Fixed-Point Blockset

For Use with SIMULINK®

Modeling

Simulation

Implementation

The MATH WORKS

User's Guide

Version 3

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Fixed-Point Blockset User's Guide

© COPYRIGHT 1995 - 2001 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and Target Language Compiler is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:

March 1995	First printing	
April 1997	Second printing	(Revised for MATLAB 5)
January 1999	Third printing	(Revised for MATLAB 5.3 (Release 11))
September 2000	Fourth printing	New for Version 3 (Release 12)
June 2001	Online only	(Revised for Version 3.1 (Release 12.1))

Contents

Preface

What Is the Fixed-Point Blockset? . . . . . **xiv**
Exploring the Blockset . . . . . **xiv**

Related Products . . . . . **xv**
System Requirements . . . . . **xv**
Associated Products . . . . . **xv**

Using This Guide . . . . . **xvii**
Expected Background . . . . . **xvii**
Learning the Fixed-Point Blockset . . . . . **xvii**
How This Book Is Organized . . . . . **xviii**

Installation Information . . . . . **xix**

Typographical Conventions . . . . . **xx**

1 Introduction

Physical Quantities and Measurement Scales **1-2**
Selecting a Measurement Scale **1-2**
Example: Selecting a Measurement Scale **1-4**

Why Use Fixed-Point Hardware? **1-9**

Why Use the Fixed-Point Blockset? **1-10**

The Development Cycle **1-11**

Compatibility with Simulink Blocks **1-13**
Frame-based Signals **1-17**
Matrix Signals **1-17**

How to Get Online Help **1-18**

2 Getting Started

An Overview of Blockset Features **2-2**
Configuring Fixed-Point Blocks **2-2**
Additional Features and Capabilities **2-8**

Example: Converting from Doubles to Fixed-Point **2-10**
Block Descriptions **2-10**
Simulation Results **2-11**

Demos **2-15**
Basic Demos **2-15**
Advanced Demos: Filters and Systems **2-16**

3 Data Types and Scaling

Overview **3-2**

Fixed-Point Numbers **3-3**
Signed Fixed-Point Numbers **3-3**
Radix Point Interpretation **3-4**
Scaling **3-5**
Quantization **3-6**
Range and Precision **3-8**
Example: Fixed-Point Scaling **3-10**
Example: Constant Scaling for Best Precision **3-12**

Floating-Point Numbers **3-16**
Scientific Notation **3-16**
The IEEE Format **3-18**
Range and Precision **3-20**
Exceptional Arithmetic **3-22**

4 Arithmetic Operations

Overview **4-2**

Limitations on Precision **4-3**
Rounding **4-3**
Padding with Trailing Zeros **4-9**
Example: Limitations on Precision and Errors **4-10**
Example: Maximizing Precision **4-11**

Limitations on Range **4-12**
Saturation and Wrapping **4-13**
Guard Bits **4-15**
Example: Limitations on Range **4-15**

Recommendations for Arithmetic and Scaling **4-16**
Addition **4-16**
Accumulation **4-19**
Multiplication **4-20**
Gain **4-21**
Division **4-23**
Summary **4-25**

Parameter and Signal Conversions **4-26**
Parameter Conversions **4-27**
Signal Conversions **4-27**

Rules for Arithmetic Operations **4-30**
Computational Units **4-30**
Addition and Subtraction **4-30**

Multiplication 4-35
Division 4-39
Shifts 4-41

Example: Conversions and Arithmetic Operations 4-46

5 Realization Structures

Overview 5-2

Direct Form II 5-4

Series Cascade Form 5-7

Parallel Form 5-10

6 Tutorial: Feedback Controller Simulation

Overview 6-2

Simulink Model of a Feedback Design 6-3

Idealized Feedback Design 6-6

Digital Controller Realization 6-7

Simulation Results 6-9
Simulation 1: Initial Guess at Scaling 6-9
Simulation 2: Global Override 6-12
Simulation 3: Automatic Scaling 6-15
Simulation 4: Individual Override 6-18

7 Building Systems and Filters

Overview 7-2
Realizations and Data Types 7-2
Realizations and Scaling 7-2

Targeting an Embedded Processor 7-3
Size Assumptions 7-3
Operation Assumptions 7-3
Design Rules 7-4

8 Producing Lookup Table Data

Overview 8-2

Worst Case Error for a Lookup Table 8-3
Example: Square Root Function 8-3

Creating Lookup Tables for a Sine Function 8-5
Parameters for fixpt_look1_func_approx 8-5
Setting Function Parameters for the Lookup Table 8-6
Example 1: Using errmax with Unrestricted Spacing 8-7
Example 2: Using nptsmax with Unrestricted Spacing 8-9
Restricting the Spacing 8-10
Example 3: Using errmax with Even Spacing 8-11
Example 4: Using nptsmax with Even Spacing 8-12
Example 5: Using errmax with Power of Two Spacing 8-13
Example 6: Using nptsmax with Power of Two Spacing 8-15
Specifying Both errmax and nptsmax 8-16
Comparing the Examples 8-17

Summary of the Lookup Table Functions 8-19

Effect of Spacing on Speed, Error, and Memory Usage . . . 8-20
Data ROM Required 8-21

Determining Out-of-Range Inputs 8-22
Determining the Location of the Input 8-22
Performing Interpolation. 8-24
Conclusion 8-26

9 Function Reference

Overview 9-2
autofixexp 9-4
fixptbestexp 9-6
fixptbestprec 9-7
fixpt_convert 9-8
fixpt_convert_prep 9-13
fixpt_look1_func_approx 9-14
fixpt_look1_func_plot 9-22
fixpt_restore_links 9-24
float 9-25
fpupdate 9-26
fxptdlg 9-28
sfix 9-31
sfrac 9-32
showfixptsimranges 9-33
sint 9-34
ufix 9-35
ufrac 9-36
uint 9-37

10 Block Reference

The Block Reference Pages 10-2

The Block Dialog Box 10-3

Common Block Features **10-4**
Block Parameters **10-4**
Block Icon Labels **10-10**
Port Data Type Display **10-10**

The Fixed-Point Blockset Libraries **10-12**
Abs **10-22**
Accumulator **10-23**
Accumulator Resettable **10-25**
Accumulator Resettable Limited **10-27**
Bit Clear **10-29**
Bit Set **10-30**
Bitwise Operator **10-31**
Compare To Constant **10-35**
Compare To Zero **10-36**
Constant **10-37**
Conversion **10-40**
Conversion Inherited **10-42**
Cosine **10-44**
Counter Free **10-45**
Counter Limited **10-46**
Data Type Duplicate **10-47**
Data Type Propagation **10-49**
Dead Zone **10-58**
Dead Zone Dynamic **10-60**
Decrement Real World **10-61**
Decrement Stored Integer **10-62**
Decrement Time To Zero **10-63**
Decrement To Zero **10-64**
Derivative **10-65**
Detect Change **10-67**
Detect Decrease **10-68**
Detect Fall Negative **10-69**
Detect Fall Nonpositive **10-70**
Detect Increase **10-71**
Detect Rise Nonnegative **10-72**
Detect Rise Positive **10-73**
Difference **10-74**
Dot Product **10-76**
Filter Direct Form I **10-78**

Filter Direct Form I Time Varying **10-81**
Filter Direct Form II **10-83**
Filter Direct Form II Time Varying **10-85**
Filter First Order **10-87**
Filter Lead or Lag **10-89**
Filter Real Zero **10-91**
FIR **10-93**
Gain **10-97**
Gateway In **10-100**
Gateway In Inherited **10-105**
Gateway Out **10-107**
Increment Real World **10-110**
Increment Stored Integer **10-111**
Integer Delay **10-112**
Integrator Backward **10-113**
Integrator Backward Resettable **10-115**
Integrator Backward Resettable Limited **10-117**
Integrator Forward **10-120**
Integrator Forward Resettable **10-122**
Integrator Forward Resettable Limited **10-124**
Integrator Trapezoidal **10-127**
Integrator Trapezoidal Resettable **10-130**
Integrator Trapezoidal Resettable Limited **10-133**
Interval Test **10-136**
Interval Test Dynamic **10-138**
Logical Operator **10-139**
Look-Up Table **10-141**
Look-Up Table Dynamic **10-146**
Look-Up Table (2D) **10-150**
Matrix Gain **10-155**
MinMax **10-159**
MinMax Running Resettable **10-162**
MultiPort Switch **10-163**
Product **10-166**
Rate Limiter **10-169**
Rate Limiter Dynamic **10-170**
Relational Operator **10-171**
Relay **10-173**
Repeating Sequence Interpolated **10-176**
Repeating Sequence Stair **10-178**

Sample Time Multiply 10-181
Saturation 10-184
Saturation Dynamic 10-185
Scaling Strip 10-186
Sign 10-187
Sine 10-188
State-Space 10-189
Sum 10-192
Switch 10-195
Tapped Delay 10-197
Unary Minus 10-199
Unit Delay 10-200
Unit Delay Enabled 10-202
Unit Delay Enabled External IC 10-204
Unit Delay Enabled Resettable 10-206
Unit Delay Enabled Resettable External IC 10-208
Unit Delay External IC 10-210
Unit Delay Resettable 10-211
Unit Delay Resettable External IC 10-213
Unit Delay With Preview Enabled 10-215
Unit Delay With Preview Enabled Resettable 10-217
Unit Delay With Preview Enabled Resettable External RV 10-219
Unit Delay With Preview Resettable 10-222
Unit Delay With Preview Resettable External RV 10-224
Wrap To Zero 10-226
Zero-Order Hold 10-227

A Code Generation

Overview **A-2**

Code Generation Support **A-3**
Languages **A-3**
Storage Class of Variables **A-3**
Storage Class of Parameters **A-3**
Rounding Modes **A-3**

Overflow Handling A-4
Blocks A-4
Scaling A-4

Generating Pure Integer Code **A-5**
Example: Generating Pure Integer Code **A-5**

Using the Simulink Accelerator **A-11**

Using External Mode or rsim Target **A-12**
External Mode **A-12**
Rapid Simulation Target **A-12**

Customizing Generated Code **A-13**
Macros Versus Functions **A-13**
Bit Sizes for Target C Compiler **A-13**

B Selected Bibliography

Preface

What Is the Fixed-Point Blockset? xii
Exploring the Blockset xii

Related Products xiii
System Requirements xiii
Associated Products xiii

Using This Guide xv
Expected Background xv
Learning the Fixed-Point Blockset xv
How This Book Is Organized xvi

Installation Information xvii

Typographical Conventions xviii

What Is the Fixed-Point Blockset?

The Fixed-Point Blockset includes a collection of blocks that extend the standard Simulink® block library. With these blocks, you can create discrete-time dynamic systems that use fixed-point arithmetic. As a result, Simulink can simulate effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering. The Fixed-Point Blockset includes these major features:

- Integer, fractional, and generalized fixed-point data types:
  - Unsigned and two's complement formats
  - Word size from 1 to 128 bits
- Floating-point data types:
  - IEEE-style singles and doubles
  - A nonstandard IEEE-style data type, where the fraction (mantissa) can range from 1 to 52 bits and the exponent can range from 1 to 11 bits
- Methods for overflow handling, scaling, and rounding of fixed-point data types
- Tools are provided to facilitate:
  - The collection of minimum and maximum simulation values
  - The optimization of scaling parameters
  - The display of input and output signals
- With the Real-Time Workshop®, you can generate C code for execution on a fixed-point embedded processor; the generated code uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations

Exploring the Blockset

To open the Fixed-Point Blockset library, type

```
fixpt
```

at the MATLAB® command line, or right-click on the Fixed-Point Blockset listing in the Simulink Library Browser. Double-click on any block icon in the library to see its parameter dialog box. Press the **Help** button to view the HTML-based help for that block.

Related Products

System Requirements

The Fixed-Point Blockset is a multiplatform product that you install on a host computer running any of the operating systems supported by The MathWorks. The Fixed-Point Blockset requires:

- MATLAB 6.0 (Release 12)
- Simulink 4.0 (Release 12)

In addition, if you want to modify the fixed-point blocks, you need one of the C compilers supported by the `mex` utility. If you want to generate code from your fixed-point models, you must have the Real-Time Workshop. If you want to create an executable from the generated code, you must have the appropriate C compiler and linker.

For the most up-to-date information about system requirements, see the system requirements section, available in the support area of the MathWorks Web site (`http://www.mathworks.com/support`).

Associated Products

The MathWorks provides several associated products that are especially relevant to the kinds of tasks you can perform with the Fixed-Point Blockset. For more information about any of these products, see either:

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The products section of the MathWorks Web site (`http://www.mathworks.com/products`)

Note The toolboxes listed below all include functions that extend MATLAB's capabilities. The blocksets all include blocks that extend Simulink's capabilities.

Product	Description
Control System Toolbox	Tool for modeling, analyzing, and designing control systems using classical and modern techniques
DSP Blockset	Simulink block libraries for the design, simulation, and prototyping of digital signal processing systems
Real-Time Workshop	Tool that generates customizable C code from Simulink models and automatically builds programs that can run in real time in a variety of environments
Simulink	Interactive, graphical environment for modeling, simulating, and prototyping dynamic systems
Simulink Report Generator	Tool for documenting information in Simulink and Stateflow in multiple output formats
Stateflow	Tool for graphical modeling and simulation of complex control logic
Stateflow Coder	Tool for generating highly readable, efficient C code from Stateflow diagrams
xPC Target	Tool for adding I/O blocks to Simulink block diagrams and downloading the code generated by Real-Time Workshop to a second PC that runs the xPC Target real-time kernel, for rapid prototyping and hardware-in-the-loop testing of control and DSP systems

Using This Guide

This guide describes how to use the Fixed-Point Blockset to emulate fixed-point arithmetic when simulating discrete-time dynamic systems using Simulink. It contains tutorial information that describes how to use the blockset features, as well as a reference entry for each block and function in the blockset.

Expected Background

This guide assumes you are familiar with both MATLAB and Simulink. If you are new to MATLAB, you should read *Getting Started with MATLAB*. If you are new to Simulink, you should read *Using Simulink*.

You should also have a basic understanding of Boolean algebra and binary word representations.

Learning the Fixed-Point Blockset

If You Are a New User

Start with Chapter 1, “Introduction,” which describes how the Fixed-Point Blockset can help you bridge the gap between designing a dynamic system and implementing it on fixed-point digital hardware. Then read Chapter 2, “Getting Started,” which describes many Fixed-Point Blockset features and provides a simple example. After reading this chapter, you should be able to create simple fixed-point models. If you want detailed information about a specific block, refer to Chapter 10, “Block Reference.” If you want detailed information about a specific function, refer to Chapter 9, “Function Reference.”

If You Are an Experienced User

Start with Chapter 6, “Tutorial: Feedback Controller Simulation,” which describes how to simulate a fixed-point digital controller design. You should then read those parts of the guide that address the functionality that concerns you. If you want detailed information about a specific block, refer to Chapter 10, “Block Reference.” If you want detailed information about a specific function, refer to Chapter 9, “Function Reference.”

How This Book Is Organized

The organization of this guide is described below.

Chapter Name	Description
Introduction	Describes how the Fixed-Point Blockset can help you bridge the gap between designing a dynamic system and implementing it on fixed-point digital hardware
Getting Started	Shows you how to use many Fixed-Point Blockset features. After reading this chapter, you should be able to create simple fixed-point models.
Data Types and Scaling	Describes fixed-point data types, floating-point data types, and data type scaling.
Arithmetic Operations	Describes fixed-point arithmetic and its limitations.
Realization Structures	Describes how to create fixed-point realization structures using fixed-point blocks.
Tutorial: Feedback Controller Simulation	Describes how to simulate a fixed-point digital controller design.
Building Systems and Filters	Describes how to create and use fixed-point systems and filters.
Producing Lookup Table Data	Describes how to create lookup table data using the lookup table approximation functions
Function Reference	Describes MATLAB M-file scripts and functions provided with the blockset.
Block Reference	Describes each fixed-point block in detail.
Code Generation	Describes the simulation features that are available for code generation. Recommendations for producing efficient code are provided.
Selected Bibliography	Provides a selected list of references.

Installation Information

To determine if the Fixed-Point Blockset is installed on your system, type

```
ver
```

at the MATLAB command line. When you enter this command, MATLAB displays information about the version of MATLAB you are running, including a list of installed add-on products and their version numbers. Check the list to see if the Fixed-Point Blockset appears.

For information about installing the blockset, see the *MATLAB Installation Guide* for your platform.

If you experience installation difficulties and have Web access, look for the installation and license information at the MathWorks Web site (`http://www.mathworks.com/support`).

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention Used	Example
Example code	Monospace font	To assign the value 5 to A, enter `A = 5`
Function names/syntax	Monospace font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	Monospace font	MATLAB responds with
		`A =`
		`5`

Menu titles, menu items, dialog boxes, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

1

Introduction

Physical Quantities and Measurement Scales 1-2
Selecting a Measurement Scale 1-2
Example: Selecting a Measurement Scale 1-4

Why Use Fixed-Point Hardware? 1-9

Why Use the Fixed-Point Blockset? 1-10

The Development Cycle 1-11

Compatibility with Simulink Blocks 1-13
Frame-based Signals 1-17
Matrix Signals 1-17

How to Get Online Help 1-18

Physical Quantities and Measurement Scales

A measurement of a physical quantity can take many numerical forms. For example, the boiling point of water is 100 degrees Celsius, 212 degrees Fahrenheit, 373 degrees Kelvin, or 671.4 degrees Rankine. No matter what number is given, the physical quantity is exactly the same. The numbers are different because four different scales are used.

Well known standard scales like Celsius are very convenient for the exchange of information. However, there are situations where it makes sense to create and use unique nonstandard scales. These situations usually involve making the most of a limited resource.

For example, nonstandard scales allow map makers to get the maximum detail on a fixed size sheet of paper. A typical road atlas of the USA will show each state on a two-page display. The scale of inches to miles will be unique for most states. By using a large ratio of miles to inches, all of Texas can fit on two pages. Using the same scale for Rhode Island would make poor use of the page. Using a much smaller ratio of miles to inches would allow Rhode Island to be shown with the maximum possible detail.

Fitting measurements of a variable inside an embedded processor is similar to fitting a state map on a piece of paper. The map scale should allow all the boundaries of the state to fit on the page. Similarly, the binary scale for a measurement should allow the maximum and minimum possible values to "fit." The map scale should also make the most of the paper in order to get maximum detail. Similarly, the binary scale for a measurement should make the most of the processor in order to get maximum precision.

Use of standard scales for measurements has definite compatibility advantages. However, there are times when it is worthwhile to break convention and use a unique nonstandard scale. There are also occasions when a mix of uniqueness and compatibility makes sense.

Selecting a Measurement Scale

Suppose that measurements of liquid water are to be made, and suppose that these measurements must be represented using 8-bit unsigned integers. Fortunately, the temperature range of liquid water is limited. No matter what scale is used, liquid water can only go from the freezing point to the boiling point. Therefore, this range of temperatures must be captured using just the 256 possible 8-bit values: 0,1,2,...,255.

One approach to representing the temperatures is to use a standard scale. For example, the units for the integers could be Celsius. Hence, the integers 0 and 100 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives a trivial conversion from the integers to degrees Celsius. On the downside, the numbers 101 to 255 are unused. By using this standard scale, more than 60% of the number range has been wasted.

A second approach is to use a nonstandard scale. In this scale, the integers 0 and 255 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives maximum precision since there are 254 values between freezing and boiling instead of just 99. The units are roughly 0.3921568 degrees Celsius per bit so the conversion to Celsius requires division by 2.55, which is a relatively expensive operation on most fixed-point processors.

A third approach is to use a "semi-standard" scale. For example, the integers 0 and 200 could represent water at the freezing point and at the boiling point, respectively. The units for this scale are 0.5 degrees Celsius per bit. On the downside, this scale doesn't use the numbers from 201 to 255, which represents a waste of more than 21%. On the upside, this scale permits relatively easy conversion to a standard scale. The conversion to Celsius involves division by 2, which is a very easy shift operation on most processors.

Measurement Scales: Beyond Multiplication

One of the key operations in converting from one scale to another is multiplication. The preceding case study gave three examples of conversions from a quantized integer value Q to a real-world Celsius value V that involved only multiplication.

$$V = \begin{cases} \dfrac{100°C}{100 \text{ bits}} \cdot Q_1 & \text{Conversion 1} \\ \dfrac{100°C}{255 \text{ bits}} \cdot Q_2 & \text{Conversion 2} \\ \dfrac{100°C}{200 \text{ bits}} \cdot Q_3 & \text{Conversion 3} \end{cases}$$

Graphically, the conversion is a line with slope S, which must pass through the origin. A line through the origin is called a purely linear conversion. Restricting

yourself to a purely linear conversion can be very wasteful and it is often better to use the general equation of a line.

$$V = SQ + B$$

By adding a bias term *B*, greater precision can be obtained when quantizing to a limited number of bits.

The general equation of a line gives a very useful conversion to a quantized scale. However, like all quantization methods, the precision is limited and errors can be introduced by the conversion. The general equation of a line with quantization error is given by

$$V = SQ + B \pm Error$$

If the quantized value *Q* is rounded to the nearest representable number, then

$$-\frac{S}{2} \leq Error \leq \frac{S}{2}$$

That is, the amount of quantization error is determined by both the number of bits and by the scale. This scenario represents the best case error. For other rounding schemes, the error can be twice as large.

Example: Selecting a Measurement Scale

On typical electronically controlled internal combustion engines, the flow of fuel is regulated to obtain the desired ratio of air to fuel in the cylinders just prior to combustion. Therefore, knowledge of the current air flow rate is required. Some manufacturers use sensors that directly measure air flow while other manufacturers calculate air flow from measurements of related signals. The relationship of these variables is derived from the ideal gas equation. The ideal gas equation involves division by air temperature. For proper results, an absolute temperature scale such as Kelvin or Rankine must be used in the equation. However, quantization directly to an absolute temperature scale would cause needlessly large quantization errors.

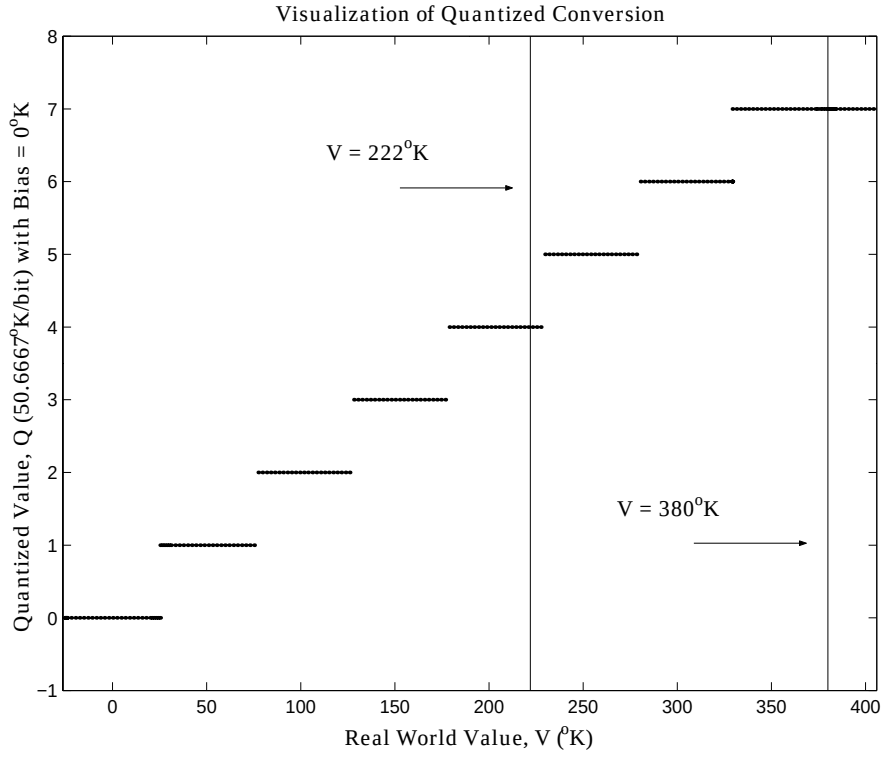
The temperature of the air flowing into the engine has a limited range. On a typical engine, the radiator is designed to keep the block below the boiling point of the cooling fluid. Let's assume a maximum of 225^{o} F (380^{o} K). As the air flows through the intake manifold, it can be heated up to this maximum temperature. For a cold start in an extreme climate, the temperature can be as

low as -60° F (222° K). Therefore, using the absolute Kelvin scale, the range of interest is 222° K to 380° K.

The air temperature needs to be quantized for processing by the embedded control system. Assuming an unrealistic quantization to 3-bit unsigned numbers: 0,1,2,...,7, the purely linear conversion with maximum precision is

$$V = \frac{380°K}{7.5 \text{ bit}} \cdot Q$$

The quantized conversion and range of interest are shown below.

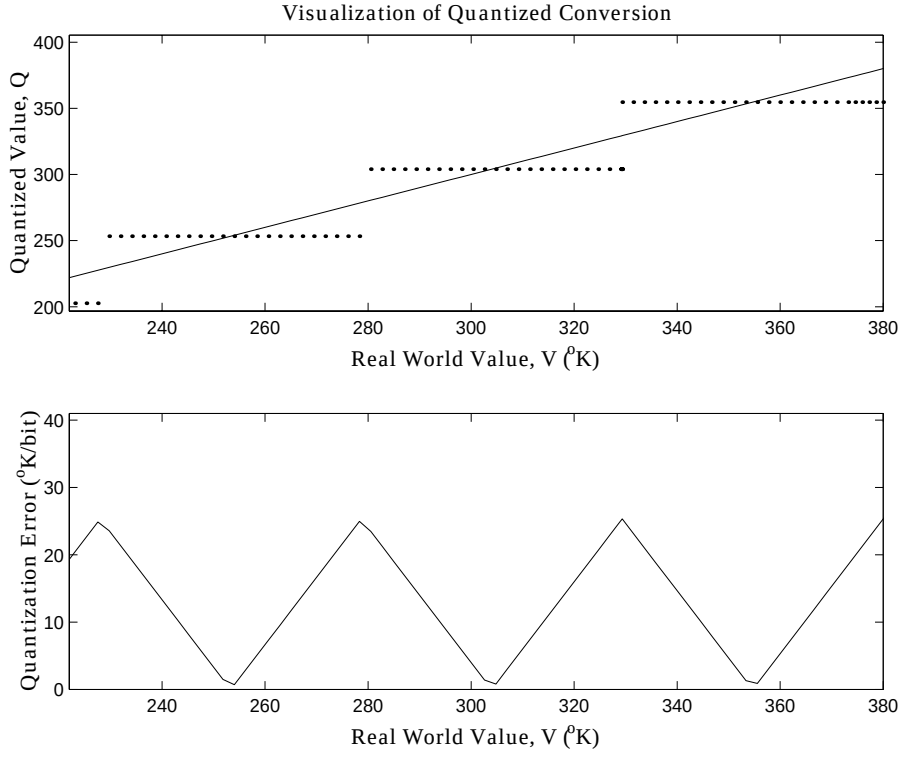


Notice that there are 7.5 possible quantization values. This is because only half of the first bit corresponds to temperatures (real-world values) greater than zero.

The quantization error is

$$-25.33°K/\text{bit} \le Error \le 25.33°K/\text{bit}$$

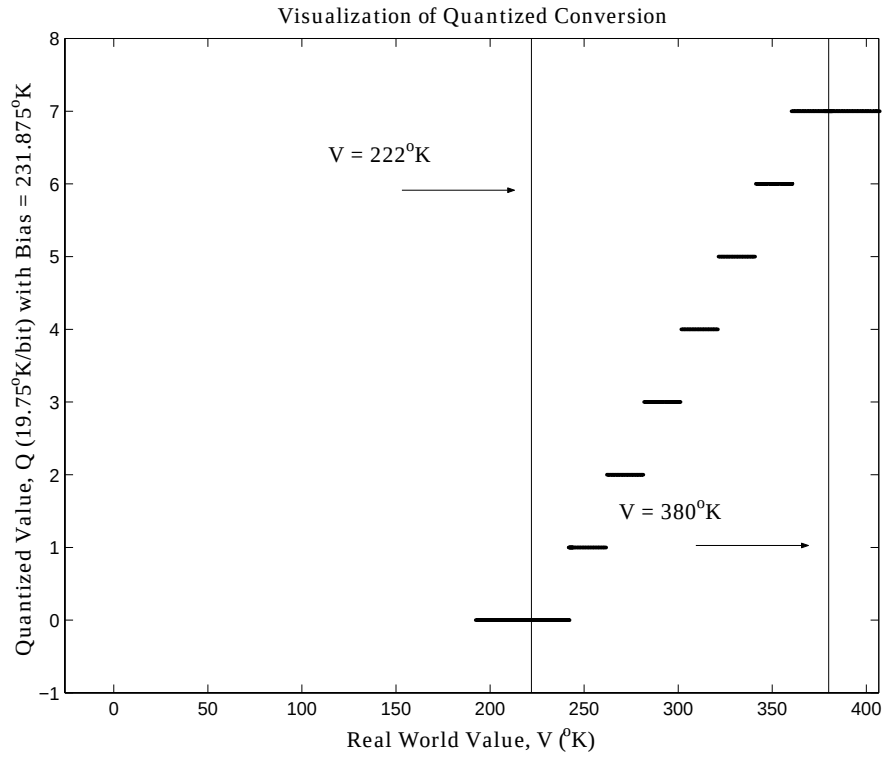
The range of interest of the quantized conversion and the absolute value of the quantized error are shown below.



As an alternative to the purely linear conversion, consider the general linear conversion with maximum precision.

$$V = \left(\frac{380°K - 222°K}{8}\right) \cdot Q + 222°K + 0.5 \cdot \left(\frac{380°K - 222°K}{8}\right)$$

The quantized conversion and range of interest are shown below.

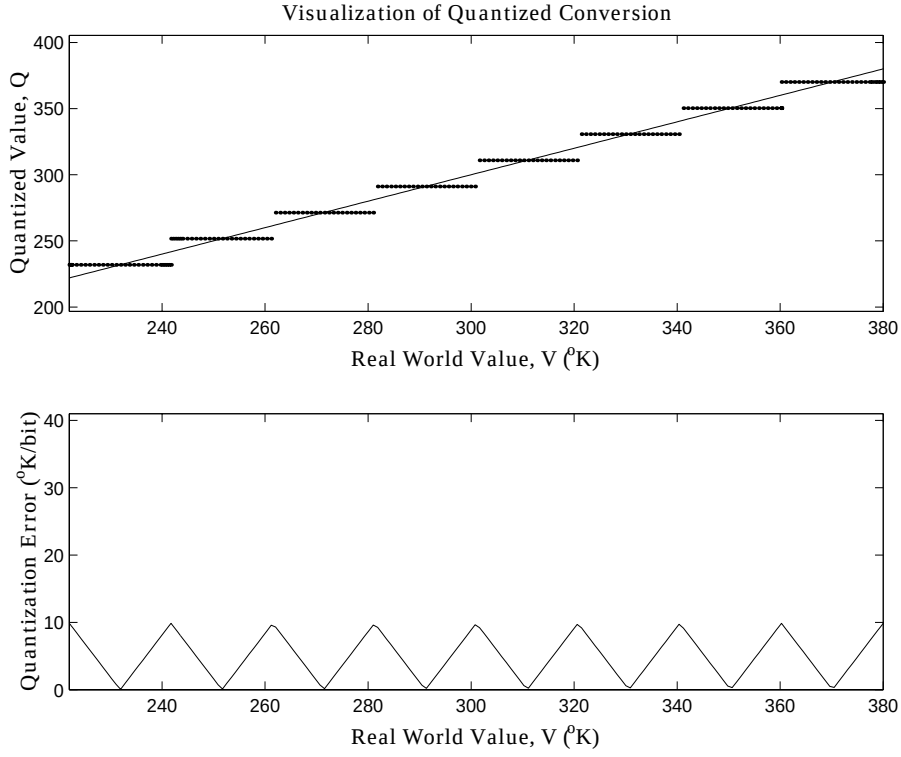


The quantization error is

$$-9.875°K/\text{bit} \leq Error \leq 9.875°K/\text{bit}$$

which is approximately 2.5 times smaller than the error associated with the purely linear conversion.

The range of interest of the quantized conversion and the absolute value of the quantized error are shown below.



Clearly, the general linear scale gives much better precision than the purely linear scale over the range of interest.

Why Use Fixed-Point Hardware?

Digital hardware is becoming the primary means in which control systems and signal processing filters are implemented. Digital hardware can be classified as either off-the-shelf hardware (for example, microcontrollers, microprocessors, general purpose processors, and digital signal processors) or custom hardware. Within these two types of hardware, there are many architecture designs. These designs range from systems with a single instruction, single data stream processing unit to systems with multiple instruction, multiple data stream processing units.

Within digital hardware, numbers are represented as either fixed-point or floating-point data types. For both these data types, word sizes are fixed at a set number of bits. However, the dynamic range of fixed-point values is much less than floating-point values with equivalent word sizes. Therefore, in order to avoid overflow or unreasonable quantization errors, fixed-point values must be scaled. Since floating-point processors can greatly simplify the real-time implementation of a control law or digital filter, and floating-point numbers can effectively approximate real-world numbers, then why use a microcontroller or processor with fixed-point hardware support? The answer to this question in many cases is cost and size:

- **Cost** – Fixed-point hardware is more cost effective where price/cost is an important consideration. When using digital hardware in a product, especially mass-produced products, fixed-point hardware, costing much less than floating-point hardware, can result in significant savings.
- **Size** – The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. For example, consider a portable telephone where one of the product design goals is to make it as portable (small and light) as possible. If one of today's high-end floating-point, general purpose processors is used, a large heat sink and battery would also be needed resulting in a costly, large, and heavy portable phone.

After making the decision to use fixed-point hardware, the next step is to choose a method for implementing the dynamic system (for example, control system or digital filter). Floating-point software emulation libraries are generally ruled out because of timing or memory size constraints. Therefore, you are left with fixed-point math where binary integer values are scaled.

Why Use the Fixed-Point Blockset?

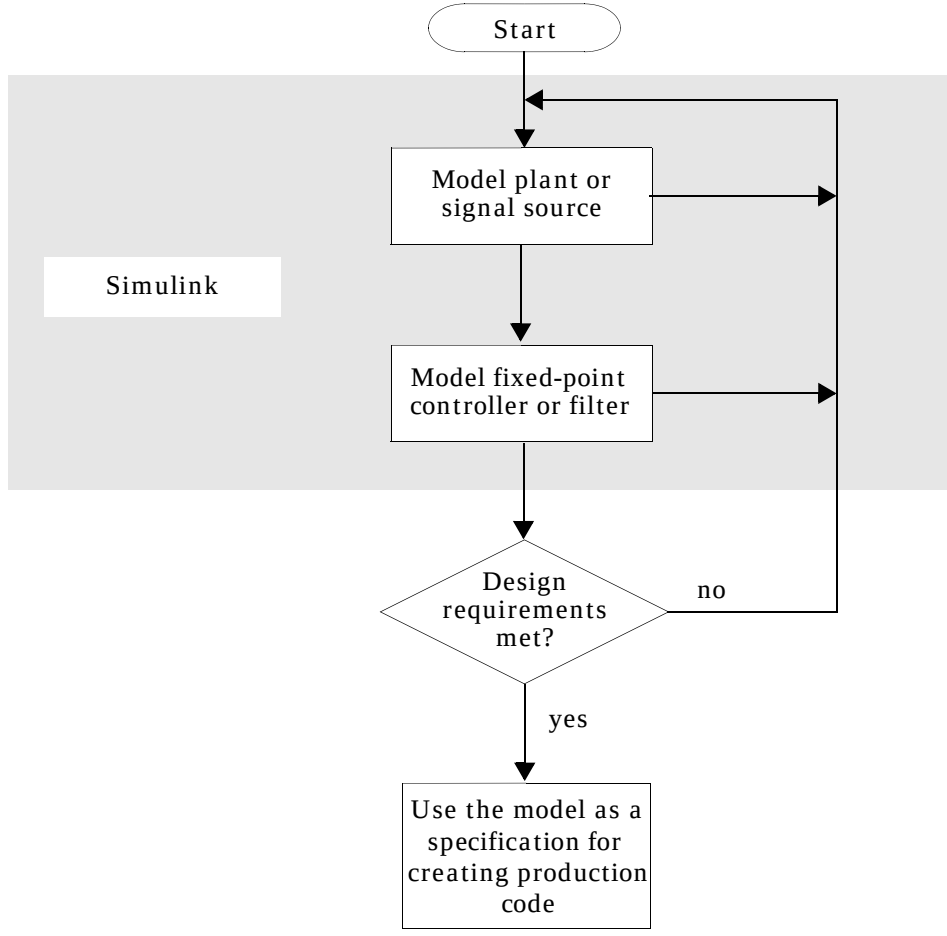
The Fixed-Point Blockset bridges the gap between designing a dynamic system and implementing it on fixed-point digital hardware. To do this, the blockset provides basic fixed-point Simulink building blocks that are used to design and simulate dynamic systems using fixed-point arithmetic. With the Fixed-Point Blockset, you can:

- Use fixed-point arithmetic to develop and simulate fixed-point Simulink models.
- Change the fixed-point data type, scaling, rounding mode, or overflow handling mode while the model is simulating. This allows you to explore issues related to numerical overflow, quantization errors, and computational noise.
- Generate fixed-point model code ready for execution on a floating-point processor. This allows you to emulate the effects of fixed-point arithmetic in a floating-point rapid prototyping system.
- Generate fixed-point model code ready for execution on a fixed-point processor.
- Modify or add new fixed-point blocks. Source code is provided for all fixed-point blocks; you will need one of the C compilers supported by the `mex` utility.

The Fixed-Point Blockset addresses the issues related to using fixed-point single instruction, single data stream processors. Extensions to multiple instruction, multiple data stream processing units can be made. However, hardware consisting of multiple instruction or multiple data streams generally also has floating-point support.

The Development Cycle

The Fixed-Point Blockset provides tools that aid in the development and testing of fixed-point dynamic systems. You directly design dynamic system models in Simulink, which are ready for implementation on fixed-point hardware. The development cycle is illustrated below.



Using MATLAB, Simulink, and the Fixed-Point Blockset, the development cycle follows these steps:

1. Model the system (plant or signal source) within Simulink using the built-in blocks and double precision numbers. Typically, the model will contain nonlinear elements.
2. Design and simulate a fixed-point dynamic system (for example, a control system or digital filter) with the Fixed-Point Blockset that meets the design, performance, and other constraints.
3. Analyze the results and go back to **1** if needed.

When the design requirements have been met, you can use the model as a specification for creating production code using the Real-Time Workshop.

The above steps interact strongly. In steps 1 and 2, there is a significant amount of freedom to select different solutions. Generally, the model is fine-tuned based upon feedback from the results of the current implementation (step 3). There is no specific modeling approach. For example, models may be obtained from first principles such as equations of motion, or from a frequency response such as a sine sweep. There are many controllers that meet the same frequency-domain or time-domain specifications. Additionally, for each controller there are an infinite number of realizations.

The Fixed-Point Blockset helps expedite the design cycle by allowing you to simulate the effects of various fixed-point controller/digital filter structures.

Compatibility with Simulink Blocks

You can connect Simulink blocks directly to Fixed-Point Blockset blocks provided the signals use built-in Simulink data types. The built-in data types include `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, `double`, and `boolean`. The Fixed-Point Blockset supports all built-in data types. However, a fixed-point signal consisting of 8-, 16-, or 32-bit integers is compatible with Simulink only when its scaling is given by a slope of `1` and a bias of `0`.

Some Simulink blocks impose restrictions on the data type of the signals they can handle. For example, some blocks accept only doubles. To incorporate these blocks into your fixed-point model, you must configure the driving block(s) to use doubles. Some Simulink blocks can accept signals of any data type. For these blocks, you can input any of the built-in data types or any of the blockset-specific data types. Examples of blockset-specific data types include 32-bit signed integers with a scaling of 2^{-8}, and 18-bit unsigned integers with a scaling of 2^{0}.

Note If you want to connect Simulink blocks to fixed-point blocks that output blockset-specific data types, then you must use the fixed-point gateway or data type conversion blocks to convert to a built-in data type.

Most Simulink blocks that accept any Fixed-Point Blockset data type have these characteristics:

- Their only function is to rearrange the signal (for example, the Selector or Mux blocks).
- They do not perform calculations such as addition or relational operations.
- They do not in general have initial conditions (for example, the Unit Delay block). However, if a Simulink block supports fixed-point, both the Simulink block and the Fixed-Point block permit the initial conditions to be set to any value.

The Simulink blocks that can accept any Fixed-Point Blockset data type are listed below.

Simulink Blocks That Accept Any Fixed-Point Data Type

Continous	Description
Memory	Output the block input from the previous time step.

Discrete	Description
Unit Delay	Sample and hold with one sample period delay.
Zero Order Hold	Implement zero-order hold of one sample period.

Math	Description
Complex to Real-Imag	Output the real and imaginary parts of a complex input signal.
Real-Imag to Complex	Output a complex signal from real and imaginary inputs.

Nonlinear	Description
Manual Switch	Switch between two inputs. (Data types must be the same.)
Multiport Switch	Choose between block inputs. (The first port cannot be fixed-point, but all others accept fixed-point)

Signals & Systems	Description
Bus Creator	Group signals into a single bus signal.
Bus Selector	Output selected input signals.
Demux	Separate a vector signal into output signals.
From	Accept input from a Goto block.
Goto	Pass block input to From blocks.
Goto Tag Visibility	Define the scope of a Goto block tag.
Ground	Gound an unconnected input port.
Matrix Concatenation	Concatenate array inputs.
Merge	Combine several input lines into a scalar line.
Mux	Combine several input lines into a vector line.
Probe	Output an input signal's width, sample time, and/or signal type.
Selector	Select or reorder the elements of the input vector.
Signal Specification	Specify attributes of a signal. (The data type must be set to auto, or use the data type propagation to force the signal to the data type.)
Terminator	Terminate an unconnected output port.
Width	Output the width of the input vector.

Subsystems	Description
Atomic Subsystem	Atomic subsystem.
Configurable Subsystem	Represent any block from a specified library.
Subsystem	Subsystem.
Enabled	Enabled subsystem.
If	If block.
If Body	A conditionally executed subsystem that can be connected to an If block.
Triggered	Triggered subsystem.
While Subsystem	While subsystem

Sinks	Description
Display	Sample and hold with one sample period delay.
Scope	Implement zero-order hold of one sample period.
To File	Write data to a file.
To Workspace	Write data to the workspace.

In some cases, fixed-point signals that are not built-in data types are converted to a real-world value as it enters the block. For example, the To Workspace block will output a 32-bit signed integer with a scaling of 2^{-8} as a double.

Refer to the *Using Simulink* guide for detailed information about the data types handled by built-in blocks.

Frame-based Signals

Signals in Simulink can be processed as frame-based signals. Most real-time systems optimize throughput rates by processing data in “batch” or “frame-based” mode, where each batch or frame is a collection of consecutive signal samples that have been buffered into a single unit.

The DSP Blockset that also supports frame-based processing, can use blocks from the Fixed-Point Blockset in models that process frame-based signals.

All Fixed-Point Blockset blocks accept frame-based signals for simulation and code generation, except for the Dot Product and FIR blocks.

For further understanding of frame-based processing, refer to Working with Signals in the DSP Blockset documentation.

Matrix Signals

The Simulink documentation refers to two-dimensional (2-D) signals as *matrices*. Simulink blocks can output two-dimensional signals where a two-dimensional (2-D) signal consists of a stream of two-dimensional arrays emitted at a frequency of one 2-D array (or matrix) per sample time.

All Fixed-Point Blockset blocks support matrix-based signals for simulation and code generation, except for the Dot Product and FIR blocks.

For further understanding of matrix-based processing, refer to Working with Signals in the Simulink documentation.

How to Get Online Help

The Fixed-Point Blockset provides several ways to get online help:

- **Block, System, and Filter Help**

  Press the **Help** button in any block, system, or filter dialog box to view its HTML-based documentation.

- **Help Desk**

  Type `helpdesk` or `doc` at the MATLAB command line to load the main MATLAB help page into the Help browser.

- **Release Information**

  Type `whatsnew fixpoint` at the MATLAB command line to view information related to the version of the Fixed-Point Blockset that you're using.

2

Getting Started

An Overview of Blockset Features 2-2
Configuring Fixed-Point Blocks 2-2
Additional Features and Capabilities 2-8

Example: Converting from Doubles to Fixed-Point . . 2-10
Block Descriptions 2-10
Simulation Results 2-11

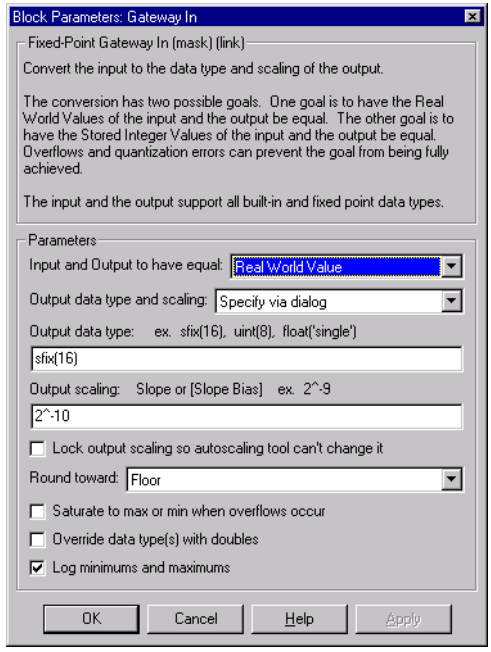
Demos 2-15
Basic Demos 2-15
Advanced Demos: Filters and Systems 2-16

An Overview of Blockset Features

This section provides a brief overview of the most important Fixed-Point Blockset features. After reading this section and "Example: Converting from Doubles to Fixed-Point" on page 2-10, you should be able to configure simple fixed-point models that suit your own application needs.

Configuring Fixed-Point Blocks

You configure fixed-point blocks with a parameter dialog box. Block configuration consists of supplying values for parameters via editable text fields, check boxes, and parameter lists. The dialog box for the Gateway In block is shown below.



The parameters associated with this block are discussed below. For detailed information about each fixed-point block, refer to Chapter 10, "Block Reference."

Real-World Values Versus Integer Values

You can configure the fixed-point gateway blocks to treat signals as real-world values or as stored integers with the **Treat input as** parameter list. The possible values are `Real World Value` and `Stored Integer`.

In terms of the variables defined in "The General Slope/Bias Encoding Scheme" on page 2-5, the real-world value is given by *V* and the stored integer value is given by *Q*. You may want to treat numbers as stored integer values if you are modeling hardware that produces integers as output.

Selecting the Output Data Type

For many fixed-point blocks, you have the option of specifying the output data type via the block dialog box, or inheriting the output data type from another block. You control how the output data type is selected with the **Output data type and scaling** parameter list. The possible values are `Specify via dialog`, `Inherit via internal rule`, and `Inherit via back propagation`. Some blocks support only two of these values.

The Fixed-Point Blockset supports several fixed-point and floating-point data types. Fixed-point data types are characterized by their word size in bits and radix (binary) point. The radix point is the means by which fixed-point values are scaled. Additionally:

- Unsigned and two's complement formats are supported.
- The fixed-point word size can range from 1 to 128 bits.
- The radix point is not required to be contiguous with the fixed-point word.

Floating-point data types are characterized by their sign bit, fraction (mantissa) field, and exponent field. The Fixed-Point Blockset supports IEEE singles, IEEE doubles, and a nonstandard IEEE-style floating-point data type.

Note You can create Fixed-Point Blockset data types directly in the MATLAB workspace and then pass the resulting structure to a fixed-point block, or you can specify the data type directly with the block dialog box.

Integers. You specify unsigned and signed integers with the `uint` and `sint` functions, respectively.

For example, to specify a 16-bit unsigned integer via the block dialog box, you configure the **Output data type** parameter as `uint(16)`. To specify a 16-bit signed integer, you configure the **Output data type** parameter as `sint(16)`.

For integer data types, the default radix point is assumed to lie to the right of all bits.

Fractional Numbers. You specify unsigned and signed fractional numbers with the `ufrac` and `sfrac` functions, respectively.

For example, to configure the output as a 16-bit unsigned fractional number via the block dialog box, you specify the **Output data type** parameter to be `ufrac(16)`. To configure a 16-bit signed fractional number, you specify **Output data type** to be `sfrac(16)`.

Fractional numbers are distinguished from integers by their default scaling. Whereas signed and unsigned integer data types have a default radix point to the right of all bits, unsigned fractional data types have a default radix point to the left of all bits, while signed fractional data types have a default radix point to the right of the sign bit.

Both unsigned and signed fractional data types support *guard bits*, which act to "guard" against overflow. For example, `sfrac(16,4)` specifies a 16-bit signed fractional number with 4 guard bits. The guard bits lie to the left of the default radix point.

Generalized Fixed-Point Numbers. You specify unsigned and signed generalized fixed-point numbers with the `ufix` and `sfix` functions. respectively.

For example, to configure the output as a 16-bit unsigned generalized fixed-point number via the block dialog box, you specify the **Output data type** parameter to be `ufix(16)`. To configure a 16-bit signed generalized fixed-point number, you specify **Output data type** to be `sfix(16)`.

Generalized fixed-point numbers are distinguished from integers and fractionals by the absence of a default scaling. For these data types, you must explicitly specify the scaling with the **Output scaling** dialog box parameter, or inherit the scaling from another block. Refer to "Selecting the Output Scaling" on page 2-5 for more information.

Floating-Point Numbers. The Fixed-Point Blockset supports single-precision and double-precision floating-point numbers as defined by the IEEE Standard

754-1985 for Binary Floating-Point Arithmetic. You specify floating-point numbers with the `float` function.

For example, to configure the output as a single-precision floating-point number via the block dialog box, you specify the **Output data type** parameter to be `float('single')`. To configure a double-precision floating-point number, you specify **Output data type** to be `float('double')`.

You can also specify a nonstandard floating-point number that mimics the IEEE style. For this data type, the fraction (mantissa) can range from 1 to 52 bits and the exponent can range from 1 to 11 bits. For example, to configure a nonstandard floating-point number having 32 total bits and 9 exponents bits, you specify **Output data type** to be `float(32,9)`.

Note These numbers are normalized with a hidden leading 1 for all exponents except the smallest possible exponent. However, the largest possible exponent might not be treated as a flag for infinity or NaNs.

Selecting the Output Scaling

Most data types supported by the Fixed-Point Blockset have a default scaling that you cannot change. However, for generalized fixed-point data types, you have the option of specifying the output scaling via the block dialog box, or inheriting the output scaling from another block. You control how the output scaling is selected with the **Output data type and scaling** parameter list.

The Fixed-Point Blockset supports two general scaling modes: radix point-only scaling and slope/bias scaling. In addition to these general scaling modes, the blockset provides you with additional block-specific scaling choices for constant vectors and constant matrices. These scaling choices are based on radix point-only scaling and are designed to maximize precision. Refer to “Example: Constant Scaling for Best Precision” in Chapter 3 for more information.

To help you understand the supported scaling modes, the general slope/bias encoding scheme is presented in the next section.

The General Slope/Bias Encoding Scheme. When representing an arbitrarily precise real-world value with a fixed-point number, it is often useful to define a general slope/bias encoding scheme

$$V \approx \tilde{V} = SQ + B$$

where:

- V is the real-world value.
- $\tilde{V}$ is the approximate real-world value.
- Q is an integer that encodes V.
- B is the bias.
- $S = F2^E$ is the slope.

The slope is partitioned into two components:

- 2^E specifies the radix point. E is the fixed power-of-two exponent.
- F is the fractional slope. It is normalized such that $1 \le F < 2$.

Radix Point-Only Scaling. This is "powers-of-two" scaling since it involves moving only the radix point. Radix point-only scaling does not require the radix point to be contiguous with the data word. The advantage of this scaling mode is the number of processor arithmetic operations is minimized.

You specify radix point-only scaling with the syntax `2^-E` where `E` is unrestricted. This creates a MATLAB structure with a bias $B = 0$ and a fractional slope $F = 1.0$.

For example, if you specify the value `2^-10` for the **Output scaling** parameter, then the generalized fixed-point number has a power-of-two exponent $E = -10$. This value defines the radix point location to be 10 places to the left of the least significant bit.

Slope/Bias Scaling. With this scaling mode, you can provide a slope and a bias. The advantage of slope/bias scaling is that it typically provides more efficient use of a finite number of bits.

You specify slope/bias scaling with the syntax `[slope bias]`, which creates a MATLAB structure with the given slope and bias.

For example, if you specify the value `[5/9 10]` for the **Output scaling** parameter, then the generalized fixed-point number has a slope of 5/9 and a bias of 10. The blockset would automatically store F as 1.1111 and E as -1 due to the normalization condition $1 \le F < 2$.

Rounding

You specify how fixed-point numbers are rounded with the **Round toward** parameter list. These rounding modes are supported:

- `Zero` – This mode rounds toward zero and is equivalent to MATLAB's `fix` function.
- `Nearest` – This mode rounds toward the nearest representable number, with the exact midpoint rounded toward positive infinity. Rounding toward nearest is equivalent to MATLAB's `round` function.
- `Ceiling` – This mode rounds toward positive infinity and is equivalent to MATLAB's `ceil` function.
- `Floor` – This mode rounds toward negative infinity and is equivalent to MATLAB's `floor` function.

Overflow Handling

You control how overflow conditions are handled for fixed-point operations with the **Saturate to max or min when overflows occur** check box.

If checked, then overflows saturate to either the maximum or minimum value represented by the data type. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127. If unchecked, then overflows wrap to the appropriate value that is representable by the data type. For example, the number 130 does not fit in a signed 8-bit integer, and would wrap to -126.

Locking the Output Scaling

If the output data type is a generalized fixed-point number, then you have the option of locking its scaling by checking the **Lock output scaling so autoscaling tool can't change it** check box.

When locked, the automatic scaling script `autofixexp` will not change the output scaling. Otherwise, the `autofixexp` is free to adjust the scaling.

Overriding with Doubles

By checking the **Override data type(s) with doubles** check box, you can override any data type with doubles. This feature is useful when debugging a simulation. For example, if you are simulating hardware that is constrained to output integers, you can override the constraint to determine whether the hardware warrants modification or replacement.

Logging Simulation Values

By checking the **Log minimums and maximums** check box, you can save the maximum and minimum values encountered during a simulation to the MATLAB workspace. You can then access these values with the automatic scaling script `autofixexp`.

Additional Features and Capabilities

In addition to the features described in "Configuring Fixed-Point Blocks" on page 2-2, the Fixed-Point Blockset provides you with these features and capabilities:

- An automatic scaling tool
- Code generation capabilities

Automatic Scaling

You can use the `autofixexp` script to automatically change the scaling for each block that has generalized fixed-point output and does not have its scaling locked. The script uses the maximum and minimum values logged during the last simulation run. The scaling is changed such that the simulation range is covered and the precision is maximized.

As an alternative to (and extension of) the automatic scaling script, you can use the Fixed-Point Blockset Interface tool. This tool allows you to easily control the parameters associated with automatic scaling and display the simulation results for a given model. Additionally, you can:

- Turn on or turn off logging for all blocks
- Override the output data type with doubles for all blocks
- Invoke the automatic scaling script

To learn how to use the Fixed-Point Blockset Interface tool, refer to Chapter 6, "Tutorial: Feedback Controller Simulation."

Code Generation

With the Real-Time Workshop, the Fixed-Point Blockset can generate C code. The code generated from fixed-point blocks uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations.

You can use the generated code on embedded fixed-point processors or rapid prototyping systems even if they contain a floating-point processor. The code is structured so that key operations can be readily replaced by optimized target-specific libraries that you supply. You can also use the Target Language Compiler™ to customize the generated code. Refer to Appendix A for more information about code generation using fixed-point blocks.

Example: Converting from Doubles to Fixed-Point

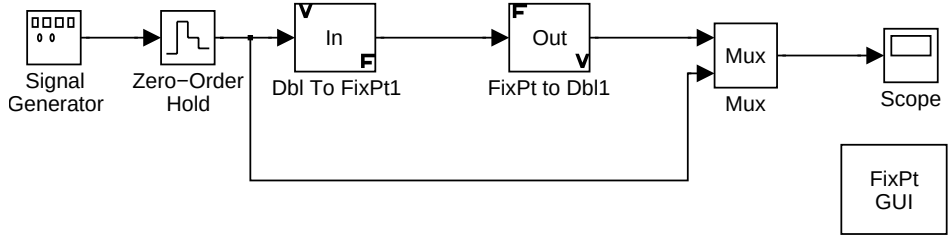
The purpose of this example is to show you how to simulate a continuous real-world signal using a generalized fixed-point data type. The model used is the simplest possible model and employs only two fixed-point blocks. Although simple in design, the model gives you the opportunity to explore many of the important features of the Fixed-Point Blockset including:

- Data types
- Scaling
- Rounding
- Logging minimum and maximum simulation values to the workspace
- Overflow handling

The model used in this example is given by the `fxpdemo_dbl2fix` demo. You can launch this demo by typing its name at the MATLAB command line.

```
fxpdemo_dbl2fix
```

The model is shown below.



Block Descriptions

The Signal Generator block is configured to output a sine wave with an amplitude defined on the interval `[-5 5]`. It always outputs double-precision numbers.

The Gateway In block is used as the interface between Simulink and the Fixed-Point Blockset. Its function is to convert the double-precision numbers from the Signal Generator block into one of the Fixed-Point Blockset data types. For simplicity, its output signal is limited to 5 bits in this example.

The Gateway Out block is used as the interface between the Fixed-Point Blockset and Simulink. Its function is to convert one of the Fixed-Point Blockset data types into a Simulink data type. In this example, it outputs double-precision numbers.

The GUI block launches the Fixed-Point Blockset Interface tool, `fxptdlg`. This tool provides convenient access to the global override and logging parameters, the logged minimum and maximum simulation data, the automatic scaling script, and the plot interface tool. It is not used in this example. However, if you have many fixed-point blocks whose scaling must be optimized, you should use this tool. Refer to Chapter 6, "Tutorial: Feedback Controller Simulation" for more information.

Note As described in "Compatibility with Simulink Blocks" on page 1-13, you can eliminate the gateway blocks from your fixed-point model if all signals use built-in data types.

Simulation Results

The results of two simulation trials are given below. The first trial uses radix point-only scaling while the second trial uses slope/bias scaling.

Trial 1: Radix Point-Only Scaling

When using radix point-only scaling, your goal is to find the optimal power-of-two exponent *E*, as defined in "Selecting the Output Scaling" on page 2-5. For this scaling mode, the fractional slope *F* is set to 1 and no bias is required.

The Gateway In block is configured in this way:

- **Output data type**

  The output data type is given by `sfix(5)`. This creates a MATLAB structure that is a 5-bit, signed generalized fixed-point number.

- **Output scaling**

  The output scaling is given by `2^–2`, which puts the radix point two places to the left of the rightmost bit. This gives a maximum value of 011.11 = 3.75, a minimum value of 100.00 = -4.00, and a precision of $(1/2)^2 = 0.25$.

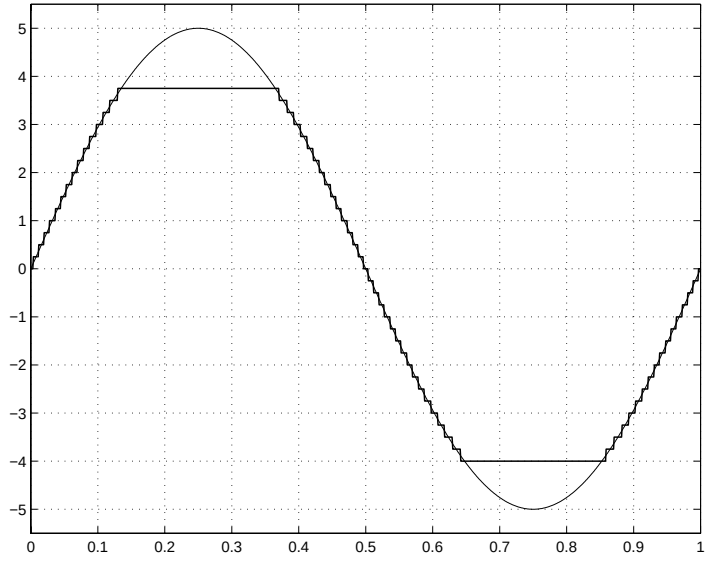
- **Rounding**

  The rounding mode is given by `Nearest`. This rounds the fixed-point result to the nearest representable number, with the exact midpoint rounded towards positive infinity.

- **Overflows**

  Fixed-point values that overflow will saturate to the maximum or minimum value represented by the word.

The resulting real-world and fixed-point simulation results are shown below.



The simulation clearly demonstrates the quantization effects of fixed-point arithmetic. The combination of using a 5-bit word with a precision of $(1/2)^2$ = 0.25 produces a discretized output that does not span the full range of the input signal.

If you want to span the complete range of the input signal with 5 bits using radix point-only scaling, then your only option is to sacrifice precision. Hence, the output scaling would be given by `2^–1`, which puts the radix point one place to the left of the rightmost bit. This scaling gives a maximum value of 0111.1 = 7.5, a minimum value of 1000.0 = -8.0, and a precision of $(1/2)^1$ = 0.5.

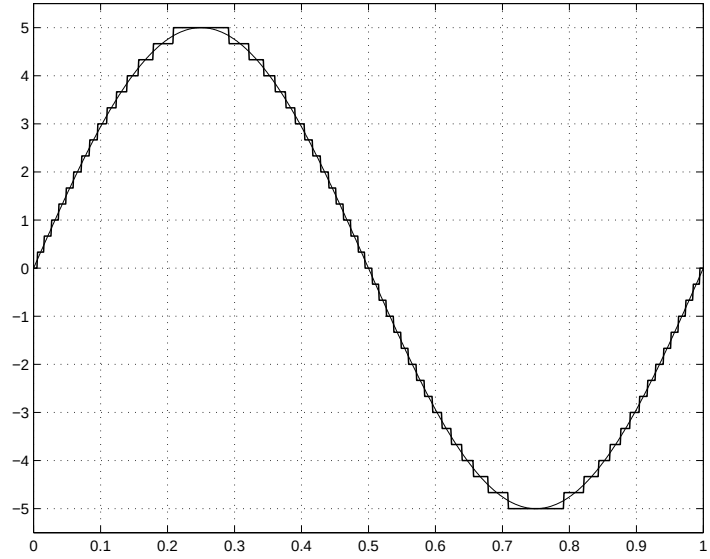
Trial 2: Slope/Bias Scaling

When using slope/bias scaling, your goal is to find the optimal fractional slope F and fixed power-of-two exponent E, as defined in "Selecting the Output Scaling" on page 2-5. No bias is required for this example since the sine wave is defined on the interval `[-5 5]`. The Gateway In block configuration is the same as that of the previous trial except for the scaling.

To arrive at a value for the slope, you can begin by assuming a fixed power-of-two exponent of -2. In the previous trial, this value defined the radix point-only scaling and resulted in a precision of 0.25. To find the fractional slope, you divide the maximum value of the sine wave by the maximum value of the scaled 5-bit number. The result is 5.00/3.75 = 1.3333. The slope (and precision) is 1.3333·(0.25) = 0.3333. You specify this value as `[0.3333]` for the **Output scaling** parameter.

Of course, you could have specified a fixed power-of-two exponent of -1 and a corresponding fractional slope of 0.6667. Naturally, the resulting slope is the same since E was reduced by one bit but F was increased by one bit. In this case, the blockset would automatically store F as 1.3332 and E as -2 due to the normalization condition of $1 \le F < 2$.

The resulting real-world and fixed-point simulation results are shown below.



This somewhat cumbersome process used to find the slope is not really necessary. All that is required is the range of the data you are simulating and the size of the fixed-point word used in the simulation. In general, you can achieve reasonable simulation results by selecting your scaling based on the formula

$$\frac{(max - min)}{2^{ws} - 1}$$

where:

- *max* is the maximum value to be simulated.
- *min* is the minimum value to be simulated.
- *ws* is the word size in bits.
- $2^{ws} - 1$ is the largest value of a word with whose size is given by *ws*.

For this example, the formula produces a slope of 0.32258.

Demos

To help you learn the Fixed-Point Blockset, a collection of demos is provided. You can explore specific blockset features by changing block parameters and observing the effects of those changes.

The demos are divided into two groups: basic demos that illustrate the basic functionality of the Fixed-Point Blockset, and advanced demos that illustrate the functionality of systems and filters built with fixed-point blocks. All demos are located in the `fxpdemos` directory.

You can access the demos through MATLAB's Demo browser. You launch the Demo browser by opening the Demos block found in the Fixed-Point Blockset library, or by typing

```
demo blockset 'Fixed Point'
```

at the command line. You can also type the name of a particular demo at the command line.

Basic Demos

The basic demos are listed below.

Table 2-1: Basic Fixed-Point Blockset Demos

Demo Name	Description
Double to Fixed-Point Conversion	Convert a double precision value to a fixed-point value.
Fixed-Point to Fixed-Point Conversion	Convert a fixed-point value to another fixed-point value.
Fixed-Point to Fixed-Point Inherited Conversion	Convert a fixed-point value to an inherited fixed-point value.
Fixed-Point Sine	Add and multiply two fixed-point sine wave signals.

Table 2-1: Basic Fixed-Point Blockset Demos (Continued)

Demo Name	Description
Scaling a Fixed-Point Control Design	Simulate a fixed-point feedback design.
Generating Only Fixed-Point Code	Generate pure integer code for a fixed-point digital controller.

The Double to Fixed-Point Conversion demo is discussed in "Example: Converting from Doubles to Fixed-Point" on page 2-10, while the Scaling a Fixed-Point Control Design demo is the subject of Chapter 6, "Tutorial: Feedback Controller Simulation."

Advanced Demos: Filters and Systems

The filter and system demos are intended to be used as a design aid so you can see how to build and test filters and systems suited to your particular needs. The output of these demos is compared to the output of analogous built-in Simulink blocks with identical input.

You can access the filter and system demos through the Filters & Systems: Examples block, which is included with the Fixed-Point Blockset library. Alternatively you can type

```
fixptsys
```

at the MATLAB command line. The advanced demos are listed below.

Table 2-2: Advanced Fixed-Point Blockset Demos

Demo Name	Description
Fixed-Point Integrators	Compare output from the Integrator Trapezoidal, Integrator Backward, and Integrator Forward blocks to output from Simulink's Discrete Integrator block.
Fixed-Point Derivatives	Compare output from the Derivative and Derivative: Filtered realizations to output from the Simulink derivatives built using the Discrete Filter and Transfer Fcn blocks.

Table 2-2: Advanced Fixed-Point Blockset Demos (Continued)

Demo Name	Description
Fixed-Point Lead and Lag Filters	Compare output from the Lead and Lag Filter block to output from analogous Simulink filters built using the Discrete Filter block.
Fixed-Point State Space	Compare output from the State-Space Realization realization to output from the analogous built-in Simulink State-Space and Discrete State-Space block.

You can invoke a filter or system demo by double-clicking the appropriate subsystem. For example, to invoke the Fixed-Point Derivatives demo, double-click the Demo: Derivative subsystem. For more information about filters and systems, refer to Chapter 7, “Building Systems and Filters.”

Additional fixed-point demos exist for direct form II, series cascade form, and parallel form realizations. These demos and realizations are discussed in Chapter 5, “Realization Structures.”

3

Data Types and Scaling

Overview 3-2

Fixed-Point Numbers 3-3
Signed Fixed-Point Numbers 3-3
Radix Point Interpretation 3-4
Scaling 3-5
Quantization 3-6
Range and Precision 3-8
Example: Fixed-Point Scaling 3-10
Example: Constant Scaling for Best Precision 3-12

Floating-Point Numbers 3-16
Scientific Notation 3-16
The IEEE Format 3-18
Range and Precision 3-20
Exceptional Arithmetic 3-22

Overview

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the radix (binary) point, and whether it is signed or unsigned. The radix point is the means by which fixed-point values are scaled. Within the Fixed-Point Blockset, fixed-point data types can be integers, fractionals, or generalized fixed-point numbers. The main difference between these data types is their default radix point. Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field. The blockset adheres to the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (referred to simply as the IEEE Standard 754 throughout this guide) and supports singles, doubles, and a nonstandard IEEE-style floating-point data type.

When choosing a data type, you must consider these factors:

- The numerical range of the result
- The precision required of the result
- The associated quantization error (i.e., the rounding mode)
- The method for dealing with exceptional arithmetic conditions

These choices depend on your specific application, the computer architecture used, and the cost of development, among others.

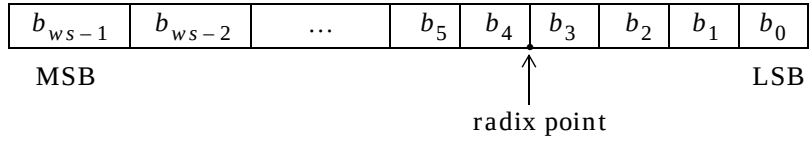
With the Fixed-Point Blockset, you can explore the relationship between data types, range, precision, and quantization error in the modeling of dynamic digital systems. With the Real-Time Workshop, you can generate production code based on that model.

Fixed-Point Numbers

Fixed-point numbers are stored in data types that are characterized by their word size in bits, radix point, and whether they are signed or unsigned. The Fixed-Point Blockset supports integers, fractionals, and generalized fixed-point numbers. The main difference between these data types is their default radix point.

Note Fixed-point word sizes up to 128 bits are supported.

A common representation of a binary fixed-point number (either signed or unsigned) is shown below.



where:

- b_i are the binary digits (bits).
- The size of the word in bits is given by *ws*.
- The most significant bit (MSB) is the leftmost bit, and is represented by location b_{ws-1}.
- The least significant bit (LSB) is the rightmost bit, and is represented by location b_0.
- The radix point is shown four places to the left of the LSB.

Signed Fixed-Point Numbers

Computer hardware typically represent the negation of a binary fixed-point number in three different ways: sign/magnitude, one's complement, and two's complement. Two's complement is the preferred representation of signed fixed-point numbers and is supported by the Fixed-Point Blockset.

Negation using two's complement consists of a bit inversion (translation into one's complement) followed by the addition of a one. For example, the two's complement of 000101 is 111011.

Whether a fixed-point value is signed or unsigned is usually not encoded explicitly within the binary word (i.e., there is no sign bit). Instead, the sign information is implicitly defined within the computer architecture.

Radix Point Interpretation

The radix point is the means by which fixed-point numbers are scaled. It is usually the software that determines the radix point. When performing basic math functions such as addition or subtraction, the hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a scale factor. They are performing signed or unsigned fixed-point binary algebra as if the radix point is to the right of b_0.

Within the Fixed-Point Blockset, the main difference between fixed-point data types is the default radix point. For integers and fractionals, the radix point is fixed at the default value. For generalized fixed-point data types, you must explicitly specify the scaling by configuring dialog box parameters, or inherit the scaling from another block. The supported fixed-point data types are described below.

Integers

The default radix point for signed and unsigned integer data types is assumed to be just to the right of the LSB. You specify unsigned and signed integers with the `uint` and `sint` functions, respectively.

Fractionals

The default radix point for unsigned fractional data types is just to the left of the MSB, while for signed fractionals the radix point is just to the right of the MSB. If you specify guard bits, then they lie to the left of the radix point. You specify unsigned and signed fractional numbers with the `ufrac` and `sfrac` functions, respectively.

Generalized Fixed-Point Numbers

For signed and unsigned generalized fixed-point numbers, there is no default radix point. You specify unsigned and signed generalized fixed-point numbers with the `ufix` and `sfix` functions. respectively.

Scaling

The dynamic range of fixed-point numbers is much less than that of floating-point numbers with equivalent word sizes. To avoid overflow conditions and minimize quantization errors, fixed-point numbers must be scaled.

With the Fixed-Point Blockset, you can select a fixed-point data type whose scaling is defined by its default radix point, or you can select a generalized fixed-point data type and choose an arbitrary linear scaling that suits your needs. This section presents the scaling choices available for generalized fixed-point data types.

A fixed-point number can be represented by a general slope/bias encoding scheme

$$V \approx \tilde{V} = SQ + B$$

where:

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value.
- Q is an integer that encodes V.
- $S = F \cdot 2^E$ is the slope.
- B is the bias.

The slope is partitioned into two components:

- 2^E specifies the radix point. E is the fixed power-of-two exponent.
- F is the fractional slope. It is normalized such that $1 \le F < 2$.

Note S and B are constants and do not show up in the computer hardware directly – only the quantization value Q is stored in computer memory.

The scaling modes available to you within this encoding scheme are described below. For detailed information about how the supported scaling modes effect fixed-point operations, refer to “Recommendations for Arithmetic and Scaling” on page 4-16.

Radix Point-Only Scaling

As the name implies, radix point-only (or "powers-of-two") scaling involves moving only the radix point within the generalized fixed-point word. The advantage of this scaling mode is the number of processor arithmetic operations is minimized.

With radix point-only scaling, the components of the general slope/bias formula have these values:

- $F = 1$
- $S = 2^E$
- $B = 0$

That is, the scaling of the quantized real-world number is defined only by the slope S, which is restricted to a power of two.

Radix point-only scaling is specified with the syntax `2^-E` where `E` is unrestricted. This creates a MATLAB structure with a bias $B = 0$ and a fractional slope $F = 1.0$. For example, the syntax `2^-10` defines a scaling such that the radix point is at a location 10 places to the left of the least significant bit.

Slope/Bias Scaling

When scaling by slope and bias, the slope S and bias B of the quantized real-world number can take on any value. Scaling by slope and bias is specified with the syntax `[slope bias]`, which creates a MATLAB structure with the given slope and bias. For example, a slope/bias scaling specified by `[5/9 10]` defines a slope of 5/9 and a bias of 10. The slope must be a positive number.

Quantization

The quantization Q of a real-world value V is represented by a weighted sum of bits. Within the context of the general slope/bias encoding scheme, the value of an unsigned fixed-point quantity is given by

$$\tilde{V} = S \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right] + B$$

while the value of a signed fixed-point quantity is given by

$$\tilde{V} = S \cdot \left[-b_{ws-1} 2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i \right] + B$$

where:

- b_i are binary digits, with $b_i = 1, 0$.
- The word size in bits is given by ws, with ws = 1,2,3,...,128.
- S is given by $F2^E$, where the scaling is unrestricted since the radix point does not have to be contiguous with the word.

b_i are called *bit multipliers* and 2^i are called the *weights*.

Example: Fixed-Point Format

The formats for 8-bit signed and unsigned fixed-point values are given below.

0	0	1	1	0	1	0	1

Unsigned data type

1	0	1	1	0	1	0	1

Signed data type

Note that you cannot discern whether these numbers are signed or unsigned data types merely by inspection since this information is not explicitly encoded within the word.

The binary number 0011.0101 yields the same value for the unsigned and two's complement representation since the MSB = 0. Setting $B = 0$ and using the appropriate weights, bit multipliers, and scaling, the value is

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right]$$

$$= 2^{-4} \cdot (0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

$$= 3.3125$$

Conversely, the binary number 1011.0101 yields different values for the unsigned and two's complement representation since the MSB = 1.

Setting $B = 0$ and using the appropriate weights, bit multipliers, and scaling, the unsigned value is

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right]$$

$$= 2^{-4} \cdot (1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

$$= 11.3125$$

while the two's complement value is

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[-b_{ws-1} 2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i \right]$$

$$= 2^{-4} \cdot (-1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

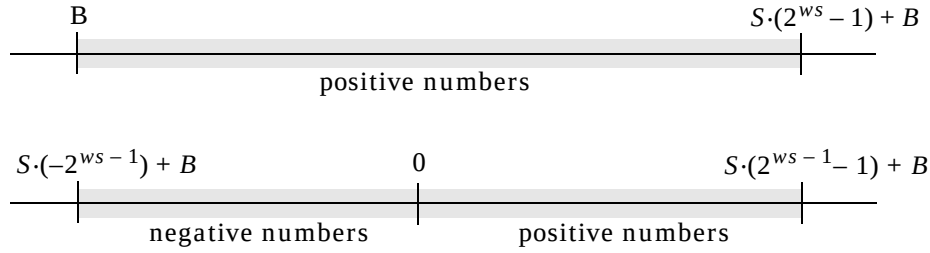
$$= -4.6875$$

Range and Precision

The *range* of a number gives the limits of the representation while the *precision* gives the distance between successive numbers in the representation. The range and precision of a fixed-point number depends on the length of the word and the scaling.

Range

The range of representable numbers for an unsigned and two's complement fixed-point number of size *ws*, scaling *S*, and bias *B* is illustrated below.



For both the signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{ws}.

For example, if the fixed-point data type is an integer with scaling defined as $S = 1$ and $B = 0$, then the maximum unsigned value is $2^{ws} - 1$ since zero must be represented. In two's complement, negative numbers must be represented as well as zero so the maximum value is $2^{ws-1} - 1$. Additionally, since there is only one representation for zero, there must be an unequal number of positive and negative numbers. This means there is a representation for -2^{ws-1} but not for 2^{ws-1}.

Precision

The precision (scaling) of integer and fractional data types is specified by the default radix point. For generalized fixed-point data types, the scaling must be explicitly defined as either slope/bias or radix point-only. In either case, the precision is given by the slope.

Fixed-Point Data Type Parameters

The low limit, high limit, and default radix point-only scaling for the supported fixed-point data types are given below.

Table 3-1: Fixed-Point Data Type Range and Default Scaling

Name	Data Type	Low Limit	High Limit	Default Scaling (~Precision)
Integer	`uint`	0	$2^{ws} - 1$	1
	`sint`	-2^{ws-1}	$2^{ws-1} - 1$	1
Fractional	`ufrac`	0	$1 - 2^{-ws}$	2^{-ws}
	`sfrac`	–1	$1 - 2^{-(ws-1)}$	$2^{-(ws-1)}$
Generalized Fixed-Point	`ufix`	N/A	N/A	N/A
	`sfix`	N/A	N/A	N/A

Example: Fixed-Point Scaling

The precision, range of signed values, and range of unsigned values for an 8-bit generalized fixed-point data type with radix point-only scaling are given below. Note that the first scaling value (2^1) represents a radix point that is not contiguous with the word.

Table 3-2: Range of an 8-Bit Fixed-Point Data Type – Radix Point-Only Scaling

Scaling	Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
2^1	2.0	-256, 254	0, 510
2^0	1.0	-128, 127	0, 255
2^{-1}	0.5	-64, 63.5	0, 127.5
2^{-2}	0.25	-32, 31.75	0, 63.75
2^{-3}	0.125	-16, 15.875	0, 31.875
2^{-4}	0.0625	-8, 7.9375	0, 15.9375

Table 3-2: Range of an 8-Bit Fixed-Point Data Type – Radix Point-Only Scaling

Scaling	Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
2^{-5}	0.03125	-4, 3.96875	0, 7.96875
2^{-6}	0.015625	-2, 1.984375	0, 3.984375
2^{-7}	0.0078125	-1, 0.9921875	0, 1.9921875
2^{-8}	0.00390625	-0.5, 0.49609375	0, 0.99609375

The precision and range of signed and unsigned values for an 8-bit fixed-point data type using slope/bias scaling are given below. The slope starts at a value of 1.25 and the bias is 1.0 for all slopes. Note that the slope is the same as the precision.

Table 3-3: Range of an 8-Bit Fixed-Point Data Type – Slope/Bias Scaling

Bias	Slope/Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
1	1.25	-159, 159.75	1, 319.75
1	0.625	-79, 80.375	1, 160.375
1	0.3125	-39, 40.6875	1, 80.6875
1	0.15625	-19, 20.84375	1, 40.84375
1	0.078125	-9, 10.921875	1, 20.921875
1	0.0390625	-4, 5.9609375	1, 10.9609375
1	0.01953125	-1.5, 3.48046875	1, 5.98046875
1	0.009765625	-0.25, 2.240234375	1, 3.490234375
1	0.0048828125	0.375, 1.6201171875	1, 2.2451171875

Example: Constant Scaling for Best Precision

The Fixed-Point Blockset provides you with block-specific modes for scaling constant vectors and constant matrices. These scaling modes are based on radix point-only scaling and are described below:

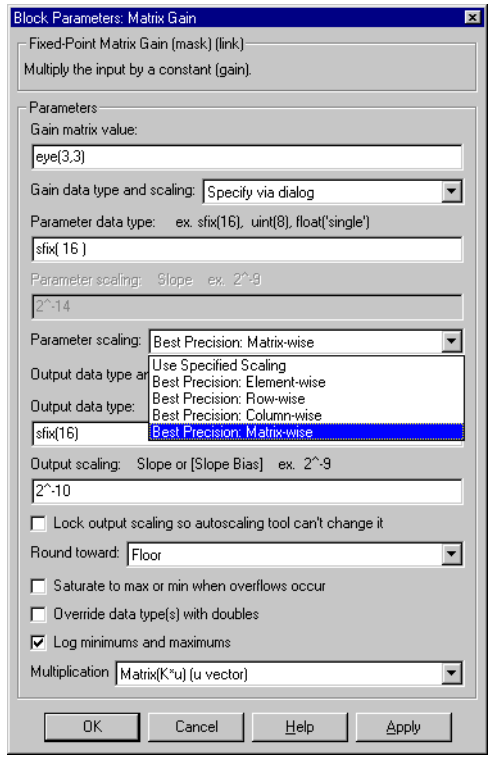
- **Constant Vector Scaling**

  Using this mode, you can scale a constant vector such that its precision is maximized element-by-element, or a common radix point is found based on the best precision for the largest value of the vector.

- **Constant Matrix Scaling**

  Using this mode, you can scale a constant matrix such that its precision is maximized element-by-element, or a common radix point is found based on the best precision for the largest value of each row, each column, or the whole matrix.

Constant matrix and constant vector scaling are available only for generalized fixed-point data types. All other fixed-point data types use their default scaling. The available constant matrix scaling modes are shown below for the Matrix Gain block.



To understand how you might use these scaling modes, consider a 5 by 4 matrix of doubles, M, defined as

```
3.3333e-005   3.3333e-006   3.3333e-007   3.3333e-008
3.3333e-004   3.3333e-005   3.3333e-006   3.3333e-007
3.3333e-003   3.3333e-004   3.3333e-005   3.3333e-006
3.3333e-002   3.3333e-003   3.3333e-004   3.3333e-005
3.3333e-001   3.3333e-002   3.3333e-003   3.3333e-004
```

Now suppose M is input into the Matrix Gain block, and you want to scale it using one of the constant matrix scaling modes. The results of using these modes are described below:

- **Use Specified Scaling**

  Suppose the matrix elements are converted to a signed, 10-bit generalized fixed-point data type with radix point-only scaling of 2^{-7} (that is, the radix

point is located seven places to the left of the rightmost bit). With this data format, M becomes

```
0              0              0              0
0              0              0              0
0              0              0              0
3.1250e-002    0              0              0
3.3594e-001    3.1250e-002    0              0
```

Note that many of the matrix elements are zero, and for the nonzero entries, the scaled values differ from the original values. This is because a double is converted to a binary word of fixed size and limited precision for each element. The larger and more precise the conversion data type, the more closely the scaled values match the original values.

- **Best Precision: Element-wise**

  If M is scaled such that the precision is maximized for each matrix element, you obtain

```
3.3379e-005   3.3304e-006   3.3341e-007   3.3295e-008
3.3379e-004   3.3379e-005   3.3304e-006   3.3341e-007
3.3340e-003   3.3379e-004   3.3379e-005   3.3304e-006
3.3325e-002   3.3340e-003   3.3379e-004   3.3379e-005
3.3301e-001   3.3325e-002   3.3340e-003   3.3379e-004
```

- **Best Precision: Row-wise**

  If M is scaled based on the largest value for each row, you obtain

```
3.3379e-005   3.3379e-006   3.5763e-007   0
3.3379e-004   3.3379e-005   2.8610e-006   0
3.3340e-003   3.3569e-004   3.0518e-005   0
3.3325e-002   3.2959e-003   3.6621e-004   0
3.3301e-001   3.3203e-002   2.9297e-003   0
```

- **Best Precision: Column-wise**

  If M is scaled based on the largest value for each column, you obtain

```
0              0              0              0
0              0              0              0
2.9297e-003    3.6621e-004    3.0518e-005    2.8610e-006
3.3203e-002    3.2959e-003    3.3569e-004    3.3379e-005
3.3301e-001    3.3325e-002    3.3340e-003    3.3379e-004
```

- **Best Precision: Matrix-wise**

  If M is scaled based on its largest matrix value, you obtain

```
0             0             0             0
0             0             0             0
2.9297e-003   0             0             0
3.3203e-002   2.9297e-003   0             0
3.3301e-001   3.3203e-002   2.9297e-003   0
```

The disadvantage of scaling the matrix column-wise, row-wise, or matrix-wise is reduced precision resulting from the use of a common radix point. The advantage of using a common radix point is reduced code size and possibly increased processor speed.

Floating-Point Numbers

Fixed-point numbers are limited in that they cannot simultaneously represent very large or very small numbers using a reasonable word size. This limitation is overcome by using scientific notation. With scientific notation, you can dynamically place the radix point at a convenient location and use powers of the radix to keep track of that location. Thus, a range of very large and very small numbers can be represented with only a few digits.

Any binary floating-point number can be represented in scientific notation form as $\pm f \times 2^{\pm e}$ where f is the fraction (or mantissa); 2 is the radix or base (binary in this case); and e is the exponent of the radix. The radix is always a positive number while f and e can be positive or negative.

When performing arithmetic operations, floating-point hardware must take into account that the sign, exponent, and fraction are all encoded within the same binary word. This results in complex logic circuits when compared with the circuits for binary fixed-point operations.

The Fixed-Point Blockset supports single-precision and double-precision floating-point numbers as defined by the IEEE Standard 754. Additionally, a nonstandard IEEE-style number is supported. To link the world of fixed-point numbers with the world of floating-point numbers, the concepts behind scientific notation are reviewed below.

Scientific Notation

A direct analogy exists between scientific notation and radix point notation. For example, scientific notation using five decimal digits for the fraction would take the form

$$\pm d.dddd \times 10^{p} = \pm ddddd.0 \times 10^{p-4} = \pm 0.ddddd \times 10^{p+1}$$

where p is an integer of unrestricted range. Radix point notation using five bits for the fraction is the same except for the number base

$$\pm b.bbbb \times 2^{q} = \pm bbbbb.0 \times 2^{q-4} = \pm 0.bbbbb \times 2^{q+1}$$

where q is an integer of unrestricted range. The previous equation is valid for both fixed- and floating-point numbers. For both these data types, the fraction can be changed at any time by the processor. However, for fixed- point numbers

the exponent never changes, while for floating-point numbers the exponent can be changed any time by the processor.

For fixed-point numbers, the exponent is fixed but there is no reason why the radix point must be contiguous with the fraction. For example, a word consisting of three unsigned bits is usually represented in scientific notation in one of these four ways.

$$bbb. = bbb. \times 2^{0}$$

$$bb.b = bbb. \times 2^{-1}$$

$$b.bb = bbb. \times 2^{-2}$$

$$.bbb = bbb. \times 2^{-3}$$

If the exponent were greater than 0 or less than -3, then the representation would involve lots of zeros.

$$bbb00000. = bbb. \times 2^{5}$$

$$bbb00. = bbb. \times 2^{2}$$

$$.00bbb = bbb. \times 2^{-5}$$

$$.00000bbb = bbb. \times 2^{-8}$$

However, these extra zeros never change to ones so they don't show up in the hardware. Furthermore, unlike floating-point exponents, a fixed-point exponent never shows up in the hardware, so fixed-point exponents are not limited by a finite number of bits.

Note The restriction of the radix point being contiguous with the fraction is unnecessary, and the Fixed-Point Blockset allows you to extend the radix point to any arbitrary location.

The IEEE Format

The IEEE Standard 754 has been widely adopted, and is used with virtually all floating-point processors and arithmetic coprocessors – with the notable exception of many DSP floating-point processors.

Among other things, this standard specifies four floating-point number formats of which singles and doubles are the most widely used. Each format contains three components: a sign bit, a fraction field, and an exponent field. These components, as well as the specific formats for singles and doubles, are discussed below.

The Sign Bit

While two's complement is the preferred representation for signed fixed-point numbers, IEEE floating-point numbers use a sign/magnitude representation, where the sign bit is explicitly included in the word. Using this representation, a sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number.

The Fraction Field

In general, floating-point numbers can be represented in many different ways by shifting the number to the left or right of the radix point and decreasing or increasing the exponent of the radix by a corresponding amount.

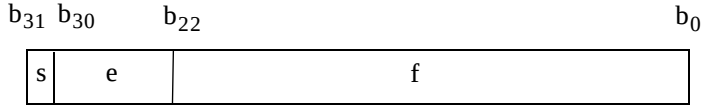
To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction of the form 1.*f* where *f* has a fixed size for a given data type. Since the leftmost fraction bit is always a 1, it is unnecessary to store this bit and is therefore implicit (or hidden). Thus, an n-bit fraction stores an n+1-bit number. The IEEE format also supports denormalized numbers, which have a fraction of the form 0.*f*. Normalized and denormalized formats are discussed in more detail in next section.

The Exponent Field

In the IEEE format, exponent representations are biased. This means a fixed value (the bias) is subtracted from the field to get the true exponent value. For example, if the exponent field is 8 bits, then the numbers 0 through 255 are represented, and there is a bias of 127. Note that some values of the exponent are reserved for flagging infinity, NaN, and denormalized numbers, so the true exponent values range from -126 to 127.

Single Precision Format

The IEEE single-precision floating-point format is a 32-bit word divided into a 1-bit sign indicator *s*, an 8-bit biased exponent *e*, and a 23-bit fraction *f*. A representation of this format is given below.



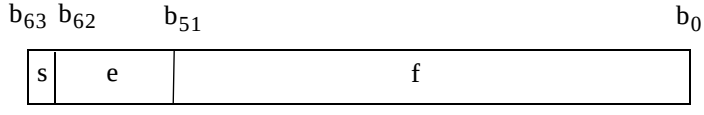
The relationship between this format and the representation of real numbers is given by

$$\text{value} = \begin{cases} (-1)^s \cdot (2^{e-127}) \cdot (1.f) & \text{normalized, } 0 < e < 255 \\ (-1)^s \cdot (2^{e-126}) \cdot (0.f) & \text{denormalized, } e = 0, \text{ f} > 0 \\ \text{exceptional value} & \text{otherwise} \end{cases}$$

Denormalized values are discussed in "Exceptional Arithmetic" on page 3-22.

Double Precision Format

The IEEE double-precision floating-point format is a 64-bit word divided into a 1-bit sign indicator *s*, an 11-bit biased exponent *e*, and a 52-bit fraction *f*. A representation of this format is given below.



The relationship between this format and the representation of real numbers is given by

$$\text{value} = \begin{cases} (-1)^s \cdot (2^{e-1023}) \cdot (1.f) & \text{normalized, } 0 < e < 2047 \\ (-1)^s \cdot (2^{e-1022}) \cdot (0.f) & \text{denormalized, } e = 0, \text{ f} > 0 \\ \text{exceptional value} & \text{otherwise} \end{cases}$$

Denormalized values are discussed in "Exceptional Arithmetic" on page 3-22.

Nonstandard IEEE Format

The Fixed-Point Blockset supports a nonstandard IEEE-style floating-point data type. This data type adheres to the definitions and formulas previously given for IEEE singles and doubles. You create nonstandard floating-point numbers with the `float` function.

```
float(TotalBits,ExpBits)
```

`TotalBits` is the total word size and `ExpBits` is the size of the exponent field. The size of the fraction field and the bias are calculated from these input arguments. You can specify any number of exponent bits up to 11, and any number of total bits such that the fraction field is no more than 53 bits.

When specifying a nonstandard format, you should remember that the number of exponent bits largely determines the range of the result and the number of fraction bits largely determines the precision of the result.

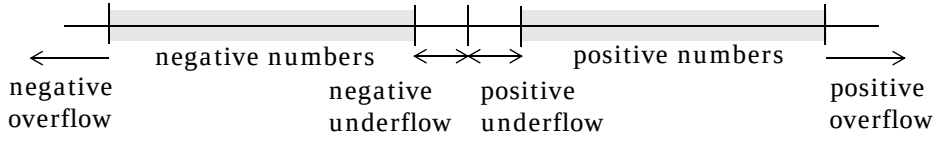
Note These numbers are normalized with a hidden leading one for all exponents except the smallest possible exponent. However, the largest possible exponent might not be treated as a flag for infinity or NaNs.

Range and Precision

The range of a number gives the limits of the representation while the precision gives the distance between successive numbers in the representation. The range and precision of an IEEE floating-point number depend on the specific format.

Range

The range of representable numbers for an IEEE floating-point number with f bits allocated for the fraction, e bits allocated for the exponent, and the bias of e given by $bias = 2^{e-1} - 1$ is given below.



where:

- Normalized positive numbers are defined within the range 2^{1-bias} to $(2-2^{-f})\cdot 2^{bias}$.
- Normalized negative numbers are defined within the range -2^{1-bias} to $-(2-2^{-f})\cdot 2^{bias}$.
- Positive numbers greater than $(2-2^{-f})\cdot 2^{bias}$, and negative numbers greater than $-(2-2^{-f})\cdot 2^{bias}$ are overflows.
- Positive numbers less than 2^{1-bias}, and negative numbers less than -2^{1-bias} are either underflows or denormalized numbers.
- Zero is given by a special bit pattern, where $e = 0$ and $f = 0$.

Overflows and underflows result from exceptional arithmetic conditions. Floating-point numbers outside the defined range are always mapped to $\pm$ inf.

Note You can use the MATLAB commands `realmin` and `realmax` to determine the dynamic range of double-precision floating-point values for your computer.

Precision

Due to a finite word size, a floating-point number is only an approximation of the "true" value. Therefore, it is important to have an understanding of the precision (or accuracy) of a floating-point result. In general, a value v with an accuracy q is specified by $v \pm q$. For IEEE floating-point numbers, $v = (-1)^s \cdot (2^{e-bias}) \cdot (1.f)$ and $q = 2^{-f} \cdot 2^{e-bias}$. Thus, the precision is associated with the number of bits in the fraction field.

Note In MATLAB, floating-point relative accuracy is given by the command `eps`, which returns the distance from 1.0 to the next largest floating point number. For a computer that supports the IEEE Standard 754, `eps` = 2^{-52} or 2.2204×10^{-16}.

Floating-Point Data Type Parameters

The high and low limits, exponent bias, and precision for the supported floating-point data types are given below.

Table 3-4: Floating-Point Data Type Parameters

Data Type	|Low Limit|	|High Limit|	Exponent Bias	Precision
single	$2^{-126} \approx 10^{-38}$	$2^{128} \approx 3 \cdot 10^{38}$	127	$2^{-23} \approx 10^{-7}$
double	$2^{-1022} \approx 2 \cdot 10^{-308}$	$2^{1024} \approx 2 \cdot 10^{308}$	1023	$2^{-52} \approx 10^{-16}$
nonstandard	$2^{(1-bias)}$	$(2-2^{-f}) \cdot 2^{bias}$	$2^{e-1}-1$	2^{-f}

Due to the sign/magnitude representation of floating-point numbers, there are two representations of zero, one positive and one negative. For both representations $e = 0$ and $0.f = 0.0$.

Exceptional Arithmetic

In addition to specifying a floating-point format, the IEEE Standard 754 specifies practices and procedures so that predictable results are produced independent of the hardware platform. Specifically, denormalized numbers, infinity, and NaNs are defined to deal with exceptional arithmetic (underflow and overflow).

If an underflow or overflow is handled as infinity or NaN, then significant processor overhead is required to deal with this exception. Although the IEEE Standard 754 specifies practices and procedures to deal with exceptional arithmetic conditions in a consistent manner, microprocessor manufacturers may handle these conditions in ways that depart from the standard. Some of the alternative approaches, such as saturation and wrapping, are discussed in Chapter 4, “Arithmetic Operations.”

Denormalized Numbers

Denormalized numbers are used to handle cases of exponent underflow. When the exponent of the result is too small (i.e., a negative exponent with too large a magnitude), the result is denormalized by right-shifting the fraction and leaving the exponent at its minimum value. The use of denormalized numbers is also referred to as gradual underflow. Without denormalized numbers, the

gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills that gap and reduces the impact of exponent underflow to a level comparable with round off among the normalized numbers. Thus, denormalized numbers provide extended range for small numbers at the expense of precision.

Infinity

Arithmetic involving infinity is treated as the limiting case of real arithmetic, with infinite values defined as those outside the range of representable numbers, or $-\infty \leq (\text{representable numbers}) < \infty$. With the exception of the special cases discussed below (NaNs), any arithmetic operation involving infinity yields infinity. Infinity is represented by the largest biased exponent allowed by the format and a fraction of zero.

NaNs

A NaN (not-a-number) is a symbolic entity encoded in floating-point format. There are two types of NaNs: signaling and quiet. A signaling NaN signals an invalid operation exception. A quiet NaN propagates through almost every arithmetic operation without signaling an exception. NaNs are produced by these operations: $\infty - \infty$, $-\infty + \infty$, $0 \times \infty$, $0/0$, and ∞/∞.

Both types of NaNs are represented by the largest biased exponent allowed by the format and a fraction that is nonzero. The bit pattern for a quiet NaN is given by $0.f$ where the most significant number in f must be a one, while the bit pattern for a signaling NaN is given by $0.f$ where the most significant number in f must be zero and at least one of the remaining numbers must be nonzero.

4

Arithmetic Operations

Overview 4-2

Limitations on Precision 4-3
Rounding 4-3
Padding with Trailing Zeros 4-9
Example: Limitations on Precision and Errors 4-10
Example: Maximizing Precision 4-11

Limitations on Range 4-12
Saturation and Wrapping 4-13
Guard Bits 4-15
Example: Limitations on Range 4-15

Recommendations for Arithmetic and Scaling 4-16
Addition 4-16
Accumulation 4-19
Multiplication 4-20
Gain 4-21
Division 4-23
Summary 4-25

Parameter and Signal Conversions 4-26
Parameter Conversions 4-27
Signal Conversions 4-27

Rules for Arithmetic Operations 4-30
Computational Units 4-30
Addition and Subtraction 4-30
Multiplication 4-35
Division 4-39
Shifts 4-41

Example: Conversions and Arithmetic Operations . . . 4-46

Overview

When developing a dynamic system using floating-point arithmetic, you generally don't have to worry about numerical limitations since floating-point data types have high precision and range. Conversely, when working with fixed-point arithmetic, you must consider these factors when developing dynamic systems:

- **Overflow**

  Adding two sufficiently large negative or positive values can produce a result that does not fit into the representation. This will have an adverse effect on the control system.

- **Quantization**

  Fixed-point values are rounded. Therefore, the output signal to the plant and the input signal to the control system do not have the same characteristics as the ideal discrete-time signal.

- **Computational noise**

  The accumulated errors that result from the rounding of individual terms within the realization introduces noise into the control signal.

- **Limit cycles**

  In the ideal system, the output of a stable transfer function (digital filter) approaches some constant for a constant input. With quantization, limit cycles occur where the output oscillates between two values in steady state.

This chapter describes the limitations involved when arithmetic operations are performed using encoded fixed-point variables. It also provides recommendations for encoding fixed-point variables such that simulations and generated code are reasonably efficient.

Limitations on Precision

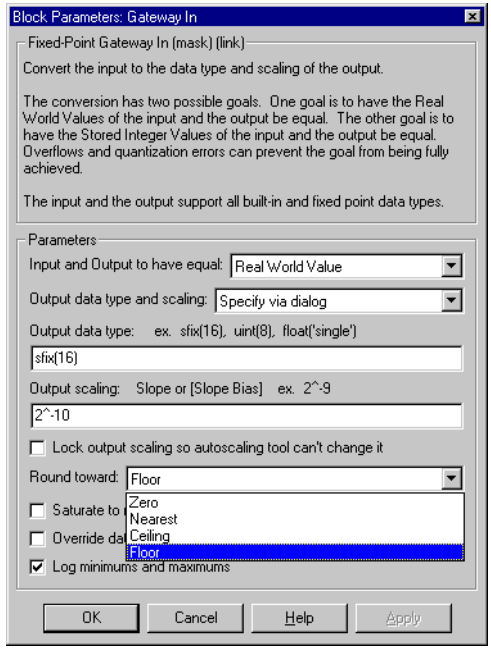
Computer words consist of a finite numbers of bits. This means that the binary encoding of variables is only an approximation of an arbitrarily precise real-world value. Therefore, the limitations of the binary representation automatically introduce limitations on the precision of the value.

The precision of a fixed-point word depends on the word size and radix point location. Extending the precision of a word can always be accomplished with more bits although you face practical limitations with this approach. Instead, you must carefully select the data type, word size, and scaling such that numbers are accurately represented. Rounding and padding with trailing zeros are typical methods implemented on processors to deal with the precision of binary words.

Rounding

The result of any operation on a fixed-point number is typically stored in a register that is longer than the number's original format. When the result is put back into the original format, the extra bits must be disposed of. That is, the result must be *rounded*. Rounding involves going from high precision to lower precision and produces quantization errors and computational noise.

The blockset provides four rounding modes, which are shown below.



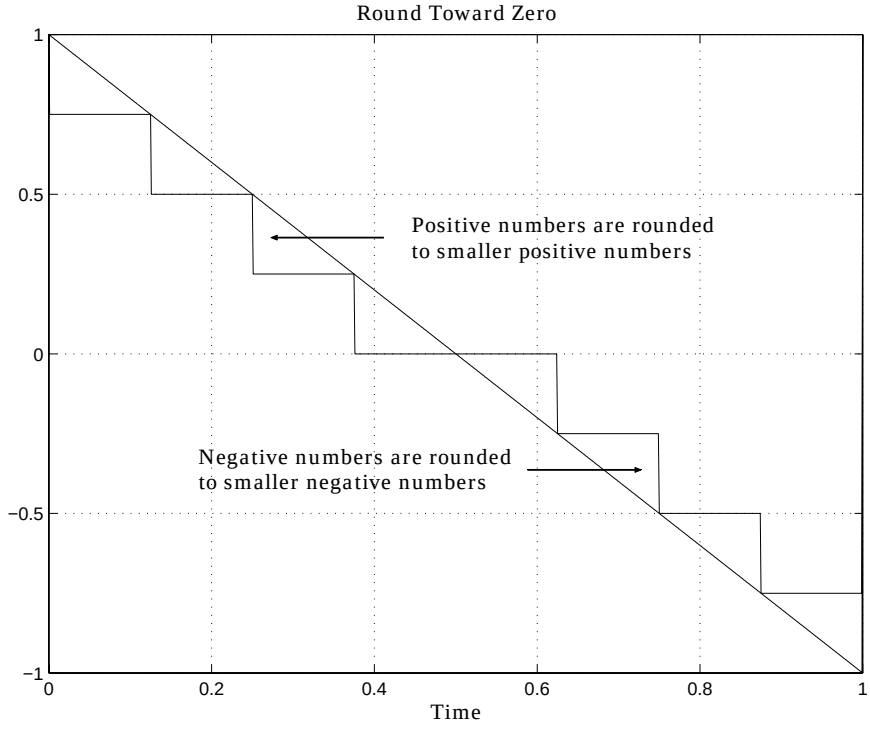
The Fixed-Point Blockset rounding modes are discussed below. The data is generated using Simulink's Signal Generator block and doubles are converted to signed 8-bit numbers with radix point-only scaling of 2^{-2}.

Round Toward Zero

The computationally simplest rounding mode is to drop all digits beyond the number required. This mode is referred to as rounding toward zero, and it results in a number whose magnitude is always less than or equal to the more precise original value. In MATLAB, you can round to zero using the `fix` function.

Rounding toward zero introduces a cumulative downward bias in the result for positive numbers and a cumulative upward bias in the result for negative numbers. That is, all positive numbers are rounded to smaller positive

numbers, while all negative numbers are rounded to smaller negative numbers. Rounding toward zero is shown below.

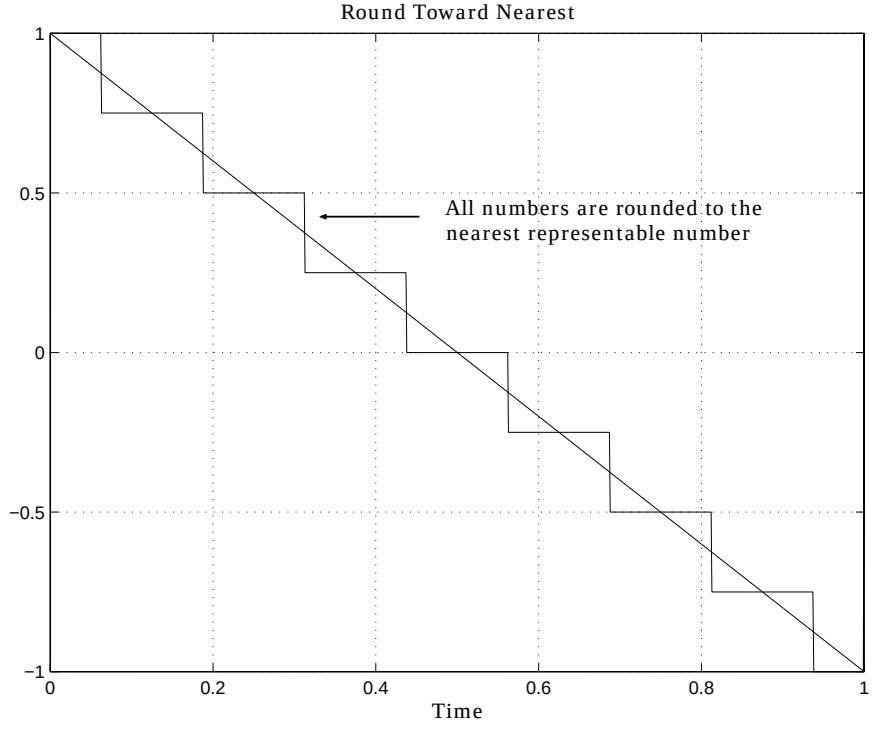


An example comparing rounding to zero and truncation for unsigned and two's complement numbers is given in "Example: Rounding to Zero Versus Truncation" on page 4-9.

Round Toward Nearest

When rounding toward nearest, the number is rounded to the nearest representable value. This mode has the smallest errors associated with it and these errors are symmetric. As a result, rounding toward nearest is the most useful approach for most applications.

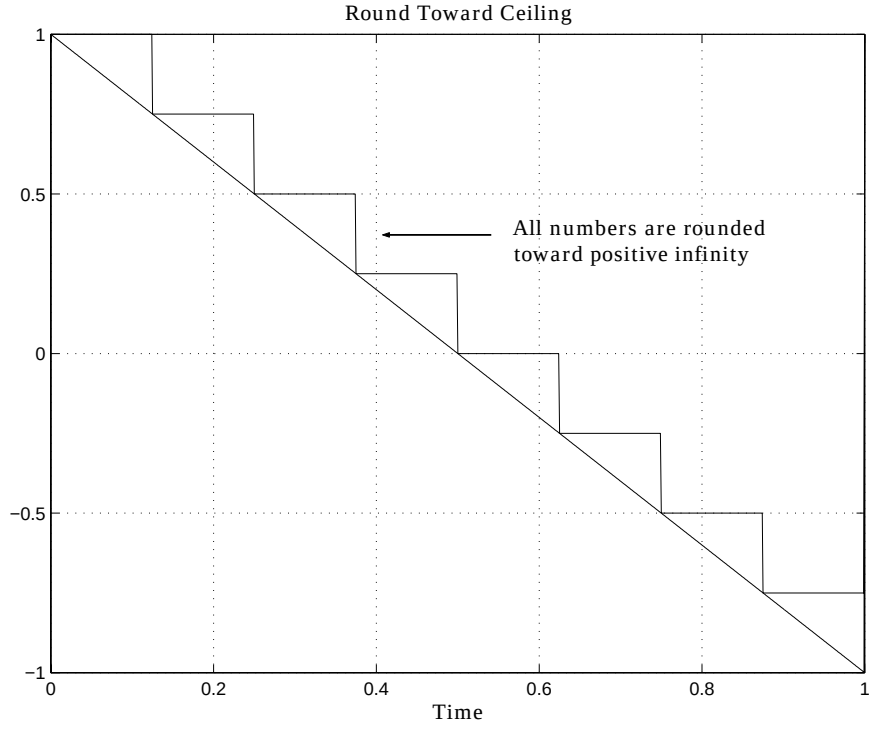
In MATLAB, you can round to nearest using the round function. Rounding toward nearest is shown below.



Round Toward Ceiling

When rounding toward ceiling, both positive and negative numbers are rounded toward positive infinity. As a result, a positive cumulative bias is introduced in the number.

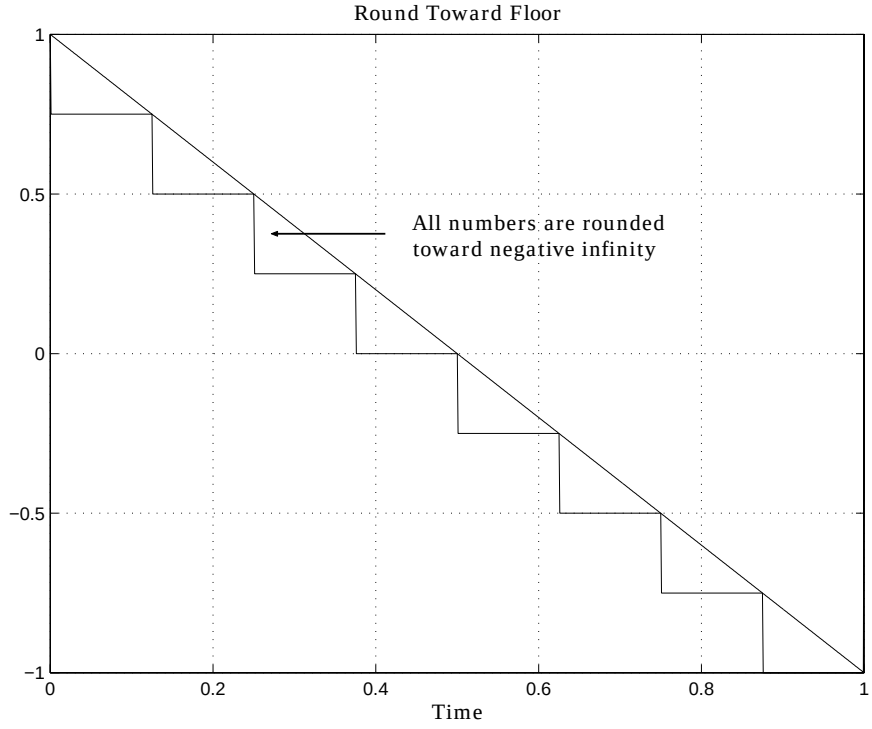
In MATLAB, you can round to ceiling using the `ceil` function. Rounding toward ceiling is shown below.



Round Toward Floor

When rounding toward floor, both positive and negative numbers are rounded to negative infinity. As a result, a negative cumulative bias is introduced in the number.

In MATLAB, you can round to floor using the `floor` function. Rounding toward floor is shown below.



Rounding toward ceiling and rounding toward floor are sometimes useful for diagnostic purposes. For example, after a series of arithmetic operations, you may not know the exact answer because of word-size limitations, which introduce rounding. If every operation in the series is performed twice, once rounding to positive infinity and once rounding to negative infinity, you obtain an upper limit and a lower limit on the correct answer. You can then decide if the result is sufficiently accurate or if additional analysis is required.

Example: Rounding to Zero Versus Truncation

Rounding to zero and *truncation* or *chopping* are sometimes thought to mean the same thing. However, the results produced by rounding to zero and truncation are different for unsigned and two's complement numbers.

To illustrate this point, consider rounding a 5-bit unsigned number to zero by dropping (truncating) the two least significant bits. For example, the unsigned number 100.01 = 4.25 is truncated to 100 = 4. Therefore, truncating an unsigned number is equivalent to rounding to zero *or* rounding to floor.

Now consider rounding a 5-bit two's complement number by dropping the two least significant bits. At first glance, you may think truncating a two's complement number is the same as rounding to zero. For example, dropping the last two digits of -3.75 yields -3.00. However, digital hardware performing two's complement arithmetic yields a different result. Specifically, the number 100.01 = -3.75 truncates to 100 = -4, which is rounding to floor.

As you can see, rounding to zero for a two's complement number is not the same as truncation when the original value is negative. For this reason, the ambiguous term "truncation" is not used in this guide, and four explicit rounding modes are used instead.

Padding with Trailing Zeros

Padding with trailing zeros involves extending the least significant bit (LSB) of a number with extra bits. This method involves going from low precision to higher precision.

For example, suppose two numbers are subtracted from each other. First, the exponents must be aligned, which typically involves a right shift of the number with the smaller value. In performing this shift, significant digits can "fall off" to the right. However, when the appropriate number of extra bits is appended, the precision of the result is maximized. Consider two 8-bit fixed-point numbers that are close in value and subtracted from each other

$$1.0000000 \cdot 2^{q} - 1.1111111 \cdot 2^{q-1}$$

where q is an integer. To perform this operation, the exponents must be equal.

$$\begin{array}{r} 1.0000000 \cdot 2^q \\ -\ 0.1111111 \cdot 2^q \\ \hline 0.0000001 \cdot 2^q \end{array}$$

If the top number is padded by two zeros and the bottom number is padded with one zero, then the above equation becomes

$$\begin{array}{r} 1.000000000 \cdot 2^q \\ -\ 0.111111110 \cdot 2^q \\ \hline 0.000000010 \cdot 2^q \end{array}$$

which produces a more precise result. An example of padding with trailing zeros using the Fixed-Point Blockset is illustrated in “Digital Controller Realization” on page 6-7.

Example: Limitations on Precision and Errors

Fixed-point variables have a limited precision because digital systems represent numbers with a finite number of bits. For example, suppose you must represent the real-world number 35.375 with a fixed-point number. Using the encoding scheme described in “Scaling” on page 3-5, the representation is

$$\tilde{V} = 2^{-2}Q + 32$$

The two closest approximations to the real-world value are $Q = 13$ and $Q = 14$.

$$\tilde{V} = 2^{-2}(13) + 32 = 35.25$$

$$\tilde{V} = 2^{-2}(14) + 32 = 35.50$$

In either case, the absolute error is the same.

$$\left|\tilde{V} - V\right| = 0.125 = \frac{F2^E}{2}$$

For fixed-point values within the limited range, this represents the worst-case error if round-to-nearest is used. If other rounding modes are used, the worst-case error can be twice as large.

$$\left|\tilde{V} - V\right| < F2^E$$

Example: Maximizing Precision

Precision is limited by slope. To achieve maximum precision, the slope should be made as small as possible while keeping the range adequately large. The bias will be adjusted in coordination with the slope.

Assume the maximum and minimum real-world value is given by $max(V)$ and $min(V)$, respectively. These limits may be known based on physical principles or engineering considerations. To maximize the precision, you must decide upon a rounding scheme and whether overflows saturate or wrap. To simplify matters, this example assumes the minimum real-world value corresponds to the minimum encoded value, and the maximum real-world value corresponds to the maximum encoded value. Using the encoding scheme described in "Scaling" on page 3-5, these values are given by

$$max(V) = F2^{E}(max(Q)) + B$$

$$min(V) = F2^{E}(min(Q)) + B$$

Solving for the slope, you get

$$F2^{E} = \frac{max(V) - min(V)}{max(Q) - min(Q)} = \frac{max(V) - min(V)}{2^{ws} - 1}$$

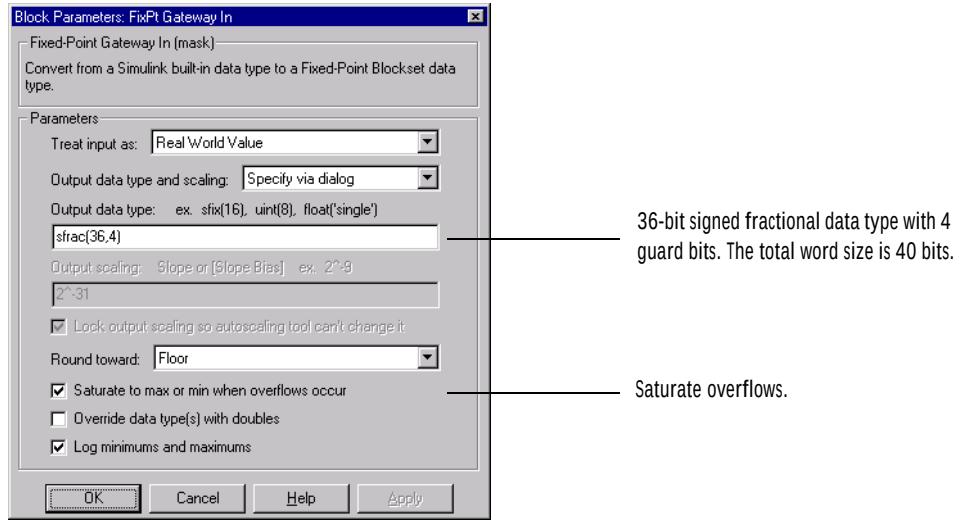
This formula is independent of rounding and overflow issues, and depends only on the word size, ws.

Limitations on Range

Limitations on the range of a fixed-point word occur for the same reason as limitations on its precision. Namely, fixed-point words have limited size.

In binary arithmetic, a processor may need to take an n-bit fixed-point number and store it in m bits, where $m \neq n$. If $m < n$, the range of the number has been reduced and an operation can produce an overflow condition. Some processors identify this condition as infinity or NaN. For other processors, especially digital signal processors (DSP's), the value *saturates* or *wraps*. If $m > n$, the range of the number has been extended. Extending the range of a word requires the inclusion of *guard bits*, which act to "guard" against potential overflow. In both cases, the range depends on the word's size and scaling.

The Fixed-Point Blockset supports saturation and wrapping for all fixed-point data types, while guard bits are supported only for fractional data types. As shown below, you can select saturation or wrapping with the **Saturate to max or min when overflows occur** check box, and you can specify guard bits with the **Output data type** parameter.

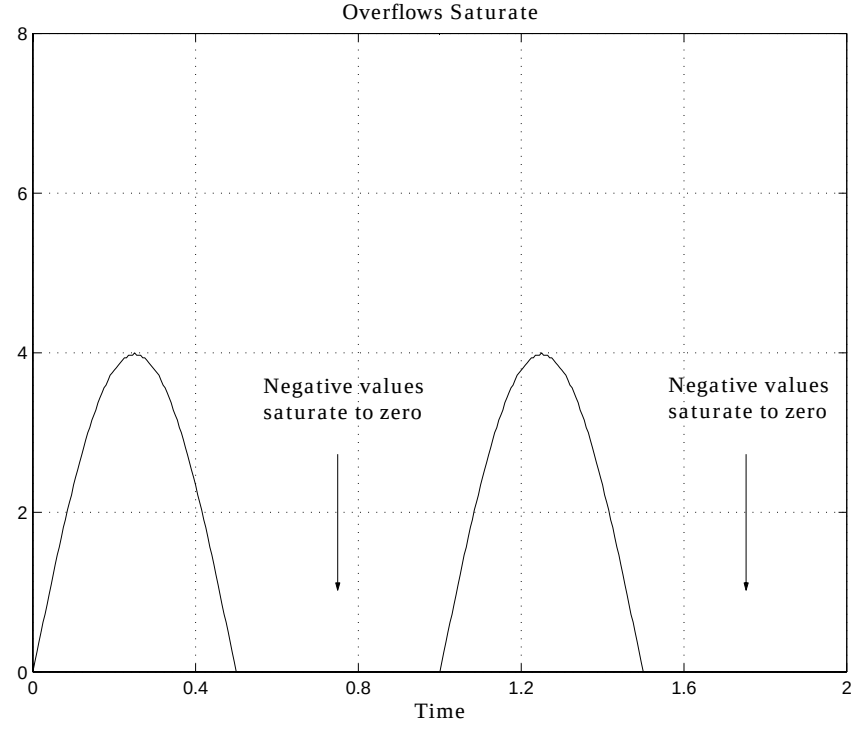


Saturation and Wrapping

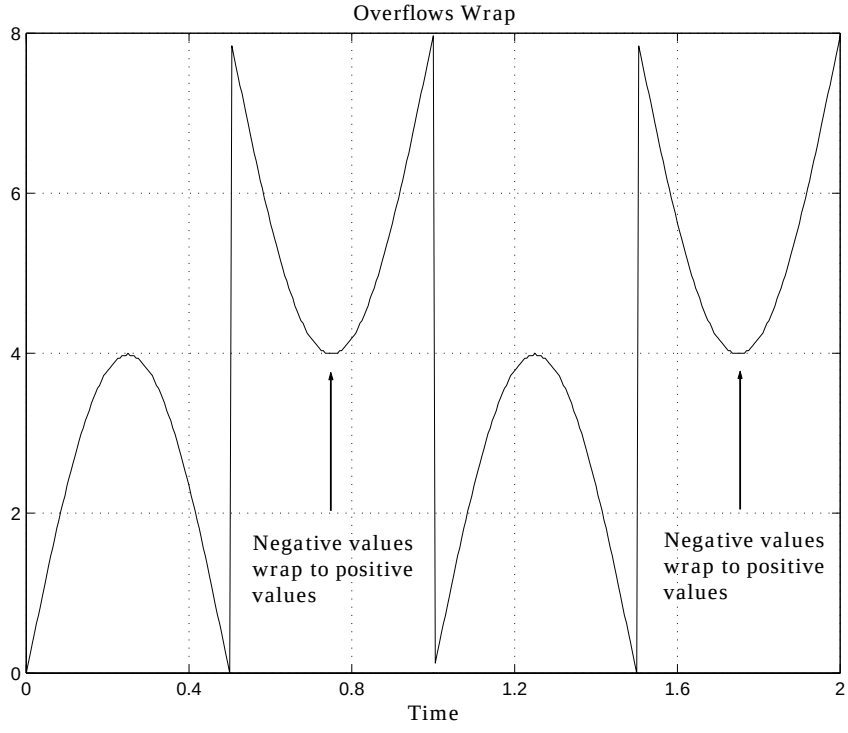
Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, Analog Device's ADSP-2100 family of processors supports either of these modes. If a register has a saturation mode of operation, then an overflow condition is set to the maximum positive or negative value allowed. Conversely, if a register has a wrapping mode of operation, an overflow condition is set to the appropriate value within the range of the representation.

Example: Saturation and Wrapping

Consider an 8-bit unsigned word with radix point-only scaling of 2^{-5}. Suppose this data type must represent a sine wave that ranges from -4 to 4. For values between 0 and 4, the word can represent these numbers without regard to overflow. This is not the case with negative numbers. If overflows saturate, all negative values are set to zero, which is the smallest number representable by the data type. The saturation of overflows is shown below.



If overflows wrap, all negative values are set to the appropriate positive value. The wrapping of overflows is shown below.



Note For most control applications, saturation is the safer way of dealing with fixed-point overflow. However, some processor architectures allow automatic saturation by hardware. If hardware saturation is not available, then extra software is required resulting in larger, slower programs. This cost is justified in some designs – perhaps for safety reasons. Other designs accept wrapping to obtain the smallest, fastest software.

Guard Bits

You can eliminate the possibility of overflow by appending the appropriate number of guard bits to a binary word.

For a two's complement signed value, the guard bits are filled with either 0's or 1's depending on the value of the most significant bit (MSB). This is called *sign extension*. For example, consider a 4-bit two's complement number with value 1011. If this number is extended in range to 7 bits with sign extension, then the number becomes 1111101 and the value remains the same.

Guard bits are supported only for fractional data types. For both signed and unsigned fractionals, the guard bits lie to the left of the default radix point.

Example: Limitations on Range

Fixed-point variables have a limited range for the same reason they have limited precision – because digital systems represent numbers with a finite number of bits. As a general example, consider the case where an integer is represented as a fixed-point word of size *ws*. The range for signed and unsigned words is given by $max(Q) - min(Q)$ where

$$min(Q) = \begin{cases} 0 & \text{unsigned} \\ -2^{ws-1} & \text{signed} \end{cases}$$

$$max(Q) = \begin{cases} 2^{ws} - 1 & \text{unsigned} \\ 2^{ws-1} - 1 & \text{signed} \end{cases}$$

Using the general slope/bias encoding scheme described in "Scaling" on page 3-5, the approximate real-world value has the range $max(\tilde{V}) - min(\tilde{V})$ where

$$min(\tilde{V}) = \begin{cases} B & \text{unsigned} \\ -F2^{E}(2^{ws-1}) + B & \text{signed} \end{cases}$$

$$max(\tilde{V}) = \begin{cases} F2^{E}(2^{ws} - 1) + B & \text{unsigned} \\ F2^{E}(2^{ws-1} - 1) + B & \text{signed} \end{cases}$$

If the real-world value exceeds the limited range of the approximate value, then the accuracy of the representation can become significantly worse.

Recommendations for Arithmetic and Scaling

This section describes the relationship between arithmetic operations and fixed-point scaling, and some basic recommendations that may be appropriate for your fixed-point design. For each arithmetic operation:

- The general slope/bias encoding scheme described in "Scaling" on page 3-5 is used.
- The scaling of the result is automatically selected based on the scaling of the two inputs. In other words, the scaling is *inherited*.
- Scaling choices are based on:
  - Minimizing the number of arithmetic operations of the result.
  - Maximizing the precision of the result.

  Additionally, radix point-only scaling is presented as a special case of the general encoding scheme.

In embedded systems, the scaling of variables at the hardware interface (the ADC or DAC) is fixed. However for most other variables, the scaling is something you can choose to give the best design. When scaling fixed-point variables, it is important to remember that:

- Your scaling choices depend on the particular design you are simulating.
- There is no best scaling approach. All choices have associated advantages and disadvantages. It is the goal of this section to expose these advantages and disadvantages to you.

Addition

Consider the addition of two real-world values.

$$V_a = V_b + V_c$$

These values are represented by the general slope/bias encoding scheme described in "Scaling" on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

In a fixed-point system, the addition of values results in finding the variable Q_a.

$$Q_a = \frac{F_b}{F_a} \cdot 2^{E_b - E_a} Q_b + \frac{F_c}{F_a} \cdot 2^{E_c - E_a} Q_c + \frac{B_b + B_c - B_a}{F_a} \cdot 2^{-E_a}$$

This formula shows:

- In general, Q_a is not computed through a simple addition of Q_b and Q_c.
- In general, there are two multiplies of a constant and a variable, two additions, and some additional bit shifting.

Inherited Scaling for Speed

In the process of finding the scaling of the sum, one reasonable goal is to simplify the calculations. Simplifying the calculations should reduce the number of operations thereby increasing execution speed. The following choices can help to minimize the number of arithmetic operations:

- Set $B_a = B_b + B_c$. This eliminates one addition.
- Set $F_a = F_b$ *or* $F_a = F_c$. Either choice eliminates one of the two constant times variable multiplies.

The resulting formula is

$$Q_a = 2^{E_b - E_a} Q_b + \frac{F_c}{F_a} \cdot 2^{E_c - E_a} Q_c$$

or

$$Q_a = \frac{F_b}{F_a} \cdot 2^{E_b - E_a} Q_b + 2^{E_c - E_a} Q_c$$

These equations appear to be equivalent. However, your choice of rounding and precision may make one choice stand out over the other. To further simplify matters, you could choose $E_a = E_c$ or $E_a = E_b$. This will eliminate some bit shifting.

Inherited Scaling for Maximum Precision

In the process of finding the scaling of the sum, one reasonable goal is maximum precision. The maximum precision scaling can be determined if the range of the variable is known. As shown in “Example: Maximizing Precision” on page 4-11, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$. For a summation, the range can be determined from

$$min(\tilde{V}_a) = min(\tilde{V}_b) + min(\tilde{V}_c)$$
$$max(\tilde{V}_a) = max(\tilde{V}_b) + max(\tilde{V}_c)$$

The maximum precision slope can now be derived.

$$F_a 2^{E_a} = \frac{max(\tilde{V}_a) - min(\tilde{V}_a)}{2^{ws_a} - 1}$$
$$= \frac{F_b 2^{E_b}(2^{ws_b} - 1) + F_c 2^{E_c}(2^{ws_c} - 1)}{2^{ws_a} - 1}$$

In most cases the input and output word sizes are much greater than one, and the slope becomes

$$F_a 2^{E_a} \approx F_b 2^{E_b + ws_b - ws_a} + F_c 2^{E_c + ws_c - ws_a}$$

which depends only on the size of the input and output words. The corresponding bias is

$$B_a = min(\tilde{V}_a) - F_a 2^{E_a} \cdot min(Q_a)$$

The value of the bias depends on whether the inputs and output are signed or unsigned numbers.

If the inputs and output are all unsigned, then the minimum value for these variables are all zero and the bias reduces to a particularly simple form.

$$B_a = B_b + B_c$$

If the inputs and the output are all signed, then the bias becomes

$$B_a \approx B_b + B_c + F_b 2^{E_b}(-2^{ws_b - 1} + 2^{ws_b - 1}) + F_c 2^{E_c}(-2^{ws_c - 1} + 2^{ws_c - 1})$$
$$B_a \approx B_b + B_c$$

Radix Point-Only Scaling

For radix point-only scaling, finding Q_a results in this simple expression.

$$Q_a = 2^{E_b - E_a} Q_b + 2^{E_c - E_a} Q_c$$

This scaling choice results in only one addition and some bit shifting. The avoidance of any multiplications is a big advantage of radix point-only scaling.

Note The subtraction of values produces results that are analogous to those produced by the addition of values.

Accumulation

The accumulation of values is closely associated with addition.

$$V_{a_new} = V_{a_old} + V_b$$

Finding Q_{a_new} involves one multiply of a constant and a variable, two additions, and some bit shifting.

$$Q_{a_new} = Q_{a_old} + \frac{F_b}{F_a} \cdot 2^{E_b - E_a} Q_b + \frac{B_b}{F_a} \cdot 2^{-E_a}$$

The important difference for fixed-point implementations is that the scaling of the output is identical to the scaling of the first input.

Radix Point-Only Scaling

For radix point-only scaling, finding Q_{a_new} results in this simple expression.

$$Q_{a_new} = Q_{a_old} + 2^{E_b - E_a} Q_b$$

This scaling option only involves one addition and some bit shifting.

Note The negative accumulation of values produces results that are analogous to those produced by the accumulation of values.

Multiplication

Consider the multiplication of two real-world values.

$$V_a = V_b \times V_c$$

These values are represented by the general slope/bias encoding scheme described in “Scaling” on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

In a fixed-point system, the multiplication of values results in finding the variable Q_a.

$$Q_a = \frac{F_b F_c}{F_a} \cdot 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} \cdot 2^{E_b - E_a} Q_b + \frac{F_c B_b}{F_a} \cdot 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} \cdot 2^{-E_a}$$

This formula shows:

- In general, Q_a is not computed through a simple multiplication of Q_b and Q_c.
- In general, there is one multiply of a constant and two variables, two multiplies of a constant and a variable, three additions, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = B_b B_c$. This eliminates one addition operation.
- Set $F_a = F_b F_c$. This simplifies the triple multiplication – certainly the most difficult part of the equation to implement.
- Set $E_a = E_b + E_c$. This eliminates some of the bit-shifting.

The resulting formula is

$$Q_a = Q_b Q_c + \frac{B_c}{F_c} \cdot 2^{-E_c} Q_b + \frac{B_b}{F_b} \cdot 2^{-E_b} Q_c$$

Inherited Scaling for Maximum Precision

The maximum precision scaling can be determined if the range of the variable is known. As shown in "Example: Maximizing Precision" on page 4-11, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$.

For multiplication, the range can be determined from

$$min(\tilde{V}_a) = min(V_{LL}, V_{LH}, V_{HL}, V_{HH})$$
$$max(\tilde{V}_a) = max(V_{LL}, V_{LH}, V_{HL}, V_{HH})$$

where

$$V_{LL} = min(\tilde{V}_b) \cdot min(\tilde{V}_c)$$
$$V_{LH} = min(\tilde{V}_b) \cdot max(\tilde{V}_c)$$
$$V_{HL} = max(\tilde{V}_b) \cdot min(\tilde{V}_c)$$
$$V_{HH} = max(\tilde{V}_b) \cdot max(\tilde{V}_c)$$

Radix Point-Only Scaling

For radix point-only scaling, finding Q_a results in this simple expression.

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c$$

Gain

Consider the multiplication of a constant and a variable

$$V_a = K \cdot V_b$$

where K is a constant called the gain. Since V_a results from the multiplication of a constant and a variable, finding Q_a is a simplified version of the general fixed-point multiply formula.

$$Q_a = \left(\frac{KF_b 2^{E_b}}{F_a 2^{E_a}} \right) \cdot Q_b + \left(\frac{KB_b - B_a}{F_a 2^{E_a}} \right)$$

Note that the terms in the parentheses can be calculated offline. Therefore, there is only one multiplication of a constant and a variable and one addition.

To implement the above equation without changing it to a more complicated form, the constants need to be encoded using a radix point-only format. For each of these constants, the range is the trivial case of only one value. Despite the trivial range, the radix point formulas for maximum precision are still valid. The maximum precision representations are the most useful choices unless there is an overriding need to avoid any shifting. The encoding of the constants is

$$\left(\frac{KF_b 2^{E_b}}{F_a 2^{E_a}} \right) = 2^{E_X} Q_X$$

$$\left(\frac{KB_b - B_a}{F_a 2^{E_a}} \right) = 2^{E_Y} Q_Y$$

resulting in the formula

$$Q_a = 2^{E_X} Q_X Q_B + 2^{E_Y} Q_Y$$

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = KB_b$. This eliminates one constant term.
- Set $F_a = KF_b$ and $E_a = E_b$. This sets the other constant term to unity.

The resulting formula is simply

$$Q_a = Q_b$$

If the number of bits is different, then either handling potential overflows or performing sign extensions is the only possible operations involved.

Inherited Scaling for Maximum Precision

The scaling for maximum precision does not need to be different than the scaling for speed unless the output has fewer bits than the input. If this is the case, then saturation should be avoided by dividing the slope by 2 for each lost bit. This will prevent saturation but will cause rounding to occur.

Division

Division of values is an operation that should be avoided in fixed-point embedded systems, but it can occur in places. Therefore, consider the division of two real-world values.

$$V_a = V_b / V_c$$

These values are represented by the general slope/bias encoding scheme described in “Scaling” on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

In a fixed-point system, the division of values results in finding the variable Q_a.

$$Q_a = \frac{F_b 2^{E_b} Q_b + B_b}{F_c F_a 2^{E_c + E_a} Q_c + B_c F_a \cdot 2^{E_a}} - \frac{B_a}{F_a} \cdot 2^{-E_a}$$

This formula shows:

- In general, Q_a is not computed through a simple division of Q_b by Q_c.
- In general, there are two multiplies of a constant and a variable, two additions, one division of a variable by a variable, one division of a constant by a variable, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = 0$. This eliminates one addition operation.
- If $B_c = 0$, then set the fractional slope $F_a = F_b / F_c$. This eliminates one constant times variable multiplication.

The resulting formula is

$$Q_a = \frac{Q_b}{Q_c} \cdot 2^{E_b - E_c - E_a} + \frac{(B_b / F_b)}{Q_c} \cdot 2^{-E_c - E_a}$$

If $B_c \neq 0$, then no clear recommendation can be made.

Inherited Scaling for Maximum Precision

The maximum precision scaling can be determined if the range of the variable is known. As shown in "Example: Maximizing Precision" on page 4-11, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$. For division, the range can be determined from

$$min(\tilde{V}_a) = min(V_{LL}, V_{LH}, V_{HL}, V_{HH})$$
$$max(\tilde{V}_a) = max(V_{LL}, V_{LH}, V_{HL}, V_{HH})$$

where for nonzero denominators

$$V_{LL} = min(\tilde{V}_b) / min(\tilde{V}_c)$$
$$V_{LH} = min(\tilde{V}_b) / max(\tilde{V}_c)$$
$$V_{HL} = max(\tilde{V}_b) / min(\tilde{V}_c)$$
$$V_{HH} = max(\tilde{V}_b) / max(\tilde{V}_c)$$

Radix Point-Only Scaling

For radix point-only scaling, finding Q_a results in this simple expression.

$$Q_a = \frac{Q_b}{Q_c} \cdot 2^{E_b - E_c - E_a}$$

Note For the last two formulas involving Q_a, a divide by zero, and zero divided by zero are possible. In these cases, the hardware will give some default behavior but you must make sure that these default responses give meaningful results for the embedded system.

Summary

From the previous analysis of fixed-point variables scaled within the general slope/bias encoding scheme, you can conclude:

- Addition, subtraction, multiplication, and division can be very involved unless certain choices are made for the biases and slopes.
- Radix point-only scaling guarantees simpler math, but generally sacrifices some precision.
- It is important to note that the previous formulas don't show that:
  - Constants and variables are represented with a finite number of bits.
  - Variables are either signed or unsigned.
  - The rounding and overflow handling schemes. These decisions must be made before an actual fixed-point realization is achieved.

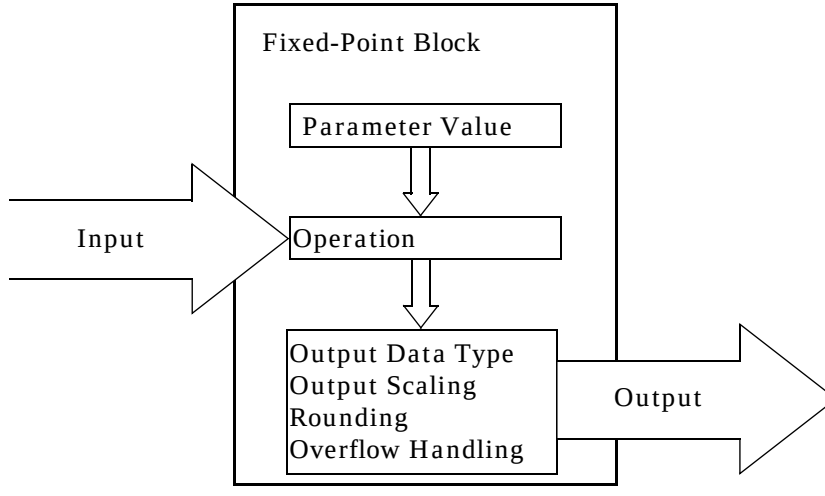
Parameter and Signal Conversions

The previous sections of this chapter, together with Chapter 3, “Data Types and Scaling,” describe how data types, scaling, rounding, overflow handling, and arithmetic operations are incorporated into the Fixed-Point Blockset. With this knowledge, you can define the output of a fixed-point model by configuring fixed-point blocks to suit your particular application.

However, to completely understand the results generated by the Fixed-Point Blockset, you must be aware of these three issues:

- When numerical block parameters are converted from a double to a Fixed-Point Blockset data type
- When input signals are converted from one Fixed-Point Blockset data type to another (if at all)
- When arithmetic operations on input signals and parameters are performed

For example, suppose a fixed-point block performs an arithmetic operation on its input signal and a parameter, and then generates output having characteristics that are specified by the block. The following diagram illustrates how these issues are related.



Parameter conversions and signal conversions are discussed below. Arithmetic operations are discussed in “Rules for Arithmetic Operations” on page 4-30.

Parameter Conversions

Block parameters that accept numerical values are always converted from a double to a Fixed-Point Blockset data type. Parameters can be converted to the input data type, the output data type, or to a data type explicitly specified by the block. For example, the FIR block converts the **Initial condition** parameter to the input data type, and converts the **FIR coefficients** parameter to a data type you explicitly specify via the block dialog box.

Parameters are always converted before any arithmetic operations are performed. Additionally, parameters are always converted *offline* using round-to-nearest and saturation. Offline conversions are discussed below.

For information about parameter conversions for a specific block, refer to Chapter 10, “Block Reference.”

Offline Conversions

An offline conversion is a conversion performed by your development platform (for example, the processor on your PC), and not by the fixed-point processor you are targeting. For example, suppose you are using a PC to develop a program to run on a fixed-point processor, and you need the fixed-point processor to compute

$$y = \left(\frac{ab}{c}\right) \cdot u = C \cdot u$$

over and over again. If *a*, *b*, and *c* are constant parameters, it is inefficient for the fixed-point processor to compute *ab*/*c* every time. Instead, the PC’s processor should compute *ab*/*c* offline one time, and the fixed-point processor computes only $C \cdot u$. This eliminates two costly fixed-point arithmetic operations.

Signal Conversions

Consider the conversion of a real-world value from one Fixed-Point Blockset data type to another. Ideally, the values before and after the conversion are equal

$$V_a = V_b$$

where V_b is the input value and V_a is the output value. To see how the conversion is implemented, the two ideal values are replaced by the general slope/bias encoding scheme described in "Scaling" on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

Solving for the output data type's stored integer value, Q_a

$$Q_a = \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b - B_a}{F_a} 2^{-E_a}$$

$$= F_s 2^{E_b - E_a} Q_b + B_{net}$$

where F_s is the adjusted fractional slope and B_{net} is the net bias. The offline conversions and online conversions and operations are discussed below.

Offline Conversions

Both F_s and B_{net} are computed offline using round-to-nearest and saturation. B_{net} is then stored using the output data type and F_s is stored using an automatically selected data type.

Online Conversions and Operations

The remaining conversions and operations are performed *online* by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The conversions and operations are given by these steps:

1 The initial value for Q_a is given by the net bias, B_{net}.

$$Q_a = B_{net}$$

2 The input integer value, Q_b, is multiplied by the adjusted slope, F_s.

$$Q_{RawProduct} = F_s Q_b$$

3 The result of step **2** is converted to the modified output data type where the slope is one and bias is zero.

$$Q_{Temp} = convert(Q_{RawProduct})$$

This conversion includes any necessary bit shifting, rounding, or overflow handling.

4 The summation operation is performed.

$$Q_a = Q_{Temp} + Q_a$$

This summation includes any necessary overflow handling.

Streamlining Simulations and Generated Code

Note that the maximum number of conversions and operations is performed when the slopes and biases of the input signal and output signal differ (are mismatched). If the scaling of these signals is identical (matched), the number of operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope and bias as the output, only step **3** is required.

$$Q_a = convert(Q_b)$$

Exclusive use of radix point-only scaling for both input signals and output signals is a common way to eliminate the occurrence of mismatched slopes and biases, and results in the most efficient simulations and generated code.

Rules for Arithmetic Operations

Fixed-point arithmetic refers to how signed or unsigned binary words are operated on. The simplicity of fixed-point arithmetic functions such as addition and subtraction allows for cost-effective hardware implementations.

This section describes the blockset-specific rules that are followed when arithmetic operations are performed on inputs and parameters. These rules are organized into four groups based on the operations involved: addition and subtraction, multiplication, division, and shifts. For each of these four groups, the rules for performing the specified operation are presented followed by an example using the rules.

Computational Units

The core architecture of many processors contains several computational units including arithmetic logic units (ALU's), multiply and accumulate units (MAC's), and shifters. These computational units process the binary data directly and provide support for arithmetic computations of varying precision. The ALU performs a standard set of arithmetic and logic operations as well as division. The MAC performs multiply, multiply/add, and multiply/subtract operations. The shifter performs logical and arithmetic shifts, normalization, denormalization, and other operations.

Addition and Subtraction

Addition is the most common arithmetic operation a processor performs. When two n-bit numbers are added together, it is always possible to produce a result with n + 1 nonzero digits due to a carry from the leftmost digit. For two's complement addition of two numbers, there are three cases to consider:

- If both numbers are positive and the result of their addition has a sign bit of 1, then overflow has occurred; otherwise the result is correct.
- If both numbers are negative and the sign of the result is 0, then overflow has occurred; otherwise the result is correct.
- If the numbers are of unlike sign, overflow cannot occur and the result is always correct.

Fixed-Point Blockset Summation Process

Consider the summation of two numbers. Ideally, the real-world values obey the equation

$$V_a = \pm V_b \pm V_c$$

where V_b and V_c are the input values and V_a is the output value. To see how the summation is actually implemented, the three ideal values should be replaced by the general slope/bias encoding scheme described in "Scaling" on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

The solution of the resulting equation for the stored integer, Q_a, is given by the equation in "Addition" on page 4-16. Using shorthand notation, that equation becomes

$$Q_a = \pm F_{sb} 2^{E_b - E_a} Q_b \pm F_{sc} 2^{E_c - E_a} Q_c + B_{net}$$

where F_{sb} and F_{sc} are the adjusted fractional slopes and B_{net} is the net bias. The offline conversions, and online conversions and operations are discussed below.

Offline Conversions. F_{sb}, F_{sc}, and B_{net} are computed offline using round-to-nearest and saturation. Furthermore, B_{net} is stored using the output data type.

Online Conversions and Operations. The remaining operations are performed online by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The worst (most inefficient) case occurs when the slopes and biases are mismatched. The worst-case conversions and operations are given by these steps:

1 The initial value for Q_a is given by the net bias, B_{net}.

$$Q_a = B_{net}$$

2 The first input integer value, Q_b, is multiplied by the adjusted slope, F_{sb}.

$$Q_{RawProduct} = F_{sb}Q_b$$

3 The previous product is converted to the modified output data type where the slope is one and the bias is zero.

$$Q_{Temp} = convert(Q_{RawProduct})$$

This conversion includes any necessary bit shifting, rounding, or overflow handling.

4 The summation operation is performed.

$$Q_a = \pm Q_a + Q_{Temp}$$

This summation includes any necessary overflow handling.

5 Steps **2** to **4** are repeated for every number to be summed.

It is important to note that bit shifting, rounding, and overflow handling are applied to the intermediate steps (**3** and **4**) and not to the overall sum.

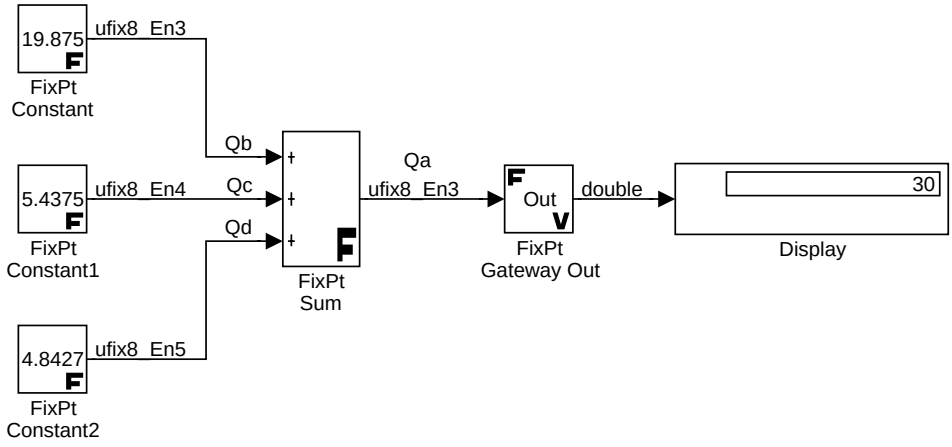
Streamlining Simulations and Generated Code

If the scaling of the input and output signals is matched, the number of summation operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope as the output, step **2** reduces to multiplication by one and can be eliminated. Trivial steps in the summation process are eliminated for both simulation and code generation. Exclusive use of radix point-only scaling for both input signals and output signals is a common way to eliminate the occurrence of mismatched slopes and biases, and results in the most efficient simulations and generated code.

Example: The Summation Process

Suppose you want to sum three numbers. Each of these numbers is represented by an 8-bit word, and each has a different radix point-only scaling. Additionally, the output is restricted to an 8-bit word with radix point-only scaling of 2^{-3}.

The summation is shown below for the input values 19.875, 5.4375, and 4.84375.



Applying the rules from the previous section, the sum follows these steps:

1 Since the biases are matched, the initial value of Q_a is trivial.

$$Q_a = 00000.000$$

2 The first number to be summed (19.875) has a fractional slope that matches the output fractional slope. Furthermore, the radix points and storage types are identical so the conversion is trivial.

$$Q_b = 10011.111$$

$$Q_{Temp} = Q_b$$

3 The summation operation is performed.

$$Q_a = Q_a + Q_{Temp} = 10011.111$$

4 The second number to be summed (5.4375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match but the difference in radix points requires that both the bits and the radix point be shifted one place to the right.

$$Q_c = 0101.0111$$

$$Q_{Temp} = convert(Q_c)$$

$$Q_{Temp} = 00101.011$$

Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow cannot occur in this case since the bits and radix point are both shifted to the right.

5 The summation operation is performed

$$Q_a = Q_a + Q_{Temp}$$

$$= \begin{array}{r} 10011.111 \\ +\ 00101.011 \\ \hline 11001.010 \end{array} = 25.250$$

Note that overflow did not occur, but it is possible for this operation.

6 The third number to be summed (4.84375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match but the difference in radix points requires that both the bits and the radix point be shifted two places to the right.

$$Q_d = 100.11011$$

$$Q_{Temp} = convert(Q_d)$$

$$Q_{Temp} = 00100.110$$

Note that a loss in precision of two bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow cannot occur in this case since the bits and radix point are both shifted to the right.

7 The summation operation is performed.

$$Q_a = Q_a + Q_{Temp}$$

$$= \begin{array}{r} 11001.010 \\ +\ 00100.110 \\ \hline 11110.000 \end{array} = 30.000$$

Note that overflow did not occur, but it is possible for this operation.

As shown below, the result of step **7** differs from the ideal sum.

$$\begin{array}{r} 10011.111 \\ 0101.0111 \\ +\quad 100.11011 \\ \hline 11110.001 \end{array} = 30.125$$

Blocks that perform addition and subtraction include the Sum, Matrix Gain, and FIR blocks.

Multiplication

The multiplication of an n-bit binary number with an m-bit binary number results in a product that is up to m + n bits in length for both signed and unsigned words. Most processors perform n-bit by n-bit multiplication and produce a 2n-bit result (double bits) assuming there is no overflow condition.

For example, the Texas Instruments TMS320C2x family of processors performs two's complement 16-bit by 16-bit multiplication and produces a 32-bit (double bit) result.

Fixed-Point Blockset Multiplication Process

Consider the multiplication of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b \times V_c$$

where V_b and V_c are the input values and V_a is the output value. To see how the multiplication is actually implemented, the three ideal values should be replaced by the general slope/bias encoding scheme described in “Scaling” on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

The solution of the resulting equation for the output stored integer, Q_a, is given below.

$$Q_a = \frac{F_b F_c}{F_a} \cdot 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} \cdot 2^{E_b - E_a} Q_b + \frac{F_c B_b}{F_a} \cdot 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} \cdot 2^{-E_a}$$

The worst-case implementation of this equation occurs when the slopes and biases of the input and output signals are mismatched. This worst-case implementation is permitted in simulation but is not always permitted for code generation since it often requires more resources than is considered practical for an embedded system. For code generation and bit-true simulations, the biases must be zero and the fractional slopes must match for most blocks. When these requirements are met, the implementation reduces to

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c$$

The bit-true implementation of this equation is discussed below.

Offline Conversions. As shown in the previous section, no offline conversions are performed.

Online Conversions and Operations. The online conversions and operations for matched slopes and biases of zero are given by these steps:

1 The integer values, Q_b and Q_c, are multiplied together.

$$Q_{RawProduct} = Q_b Q_c$$

To maintain the full precision of the product, the radix point of $Q_{RawProduct}$ is given by the sum of the radix points of Q_b and Q_c.

2 The previous product is converted to the output data type.

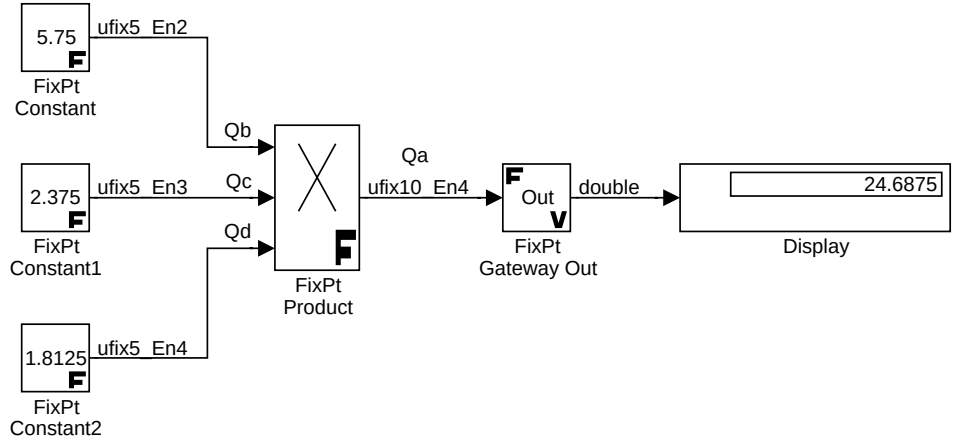
$$Q_a = convert(Q_{RawProduct})$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. Conversions are discussed in “Signal Conversions” on page 4-27.

3 Steps **1** and **2** are repeated for each additional number to be multiplied.

Example: The Multiplication Process

Suppose you want to multiply three numbers. Each of these numbers is represented by a 5-bit word, and each has a different radix point-only scaling. Additionally, the output is restricted to a 10-bit word with radix point-only scaling of 2^{-4}. The multiplication is shown below for the input values 5.75, 2.375, and 1.8125.



Applying the rules from the previous section, the multiplication follows these steps:

1 The first two numbers (5.75 and 2.375) are multiplied.

$$
\begin{aligned}
Q_{RawProduct} = \quad & 101.11 \\
& \times 10.011 \\
\hline
& 101.11 \cdot 2^{-3} \\
& 101.11 \cdot 2^{-2} \\
+ \; & 101.11 \cdot 2^{1} \\
\hline
& 01101.10101 = 13.65625
\end{aligned}
$$

Note that the radix point of the product is given by the sum of the radix points of the multiplied numbers.

2 The result of step **1** is converted to the output data type.

$$Q_{Temp} = convert(Q_{RawProduct})$$

$$= 001101.1010 = 13.6250$$

Conversions are discussed in "Signal Conversions" on page 4-27. Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

3 The result of step **2** and the third number (1.8125) are multiplied.

$$
\begin{aligned}
Q_{RawProduct} = \quad & 01101.1010 \\
& \times 1.1101 \\
\hline
& 1101.1010 \cdot 2^{-4} \\
& 1101.1010 \cdot 2^{-2} \\
& 1101.1010 \cdot 2^{-1} \\
+ \; & 1101.1010 \cdot 2^{0} \\
\hline
& 0011000.10110010 = 24.6953125
\end{aligned}
$$

Note that the radix point of the product is given by the sum of the radix points of the multiplied numbers.

4 The product is converted to the output data type.

$$Q_a = convert(Q_{RawProduct})$$

$$= 011000.1011 = 24.6875$$

Conversions are discussed in "Signal Conversions" on page 4-27. Note that a loss in precision of four bits occurred, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

Blocks that perform multiplication include the Product, FIR, Gain, and Matrix Gain blocks.

Division

As with multiplication, division with mismatched scaling is complicated. Mismatched division is permitted for simulation only. For code generation and bit-true simulation, the signals must all have zero biases and matched fractional slopes.

Fixed-Point Blockset Division Process

Consider the division of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b / V_c$$

where V_b and V_c are the input values and V_a is the output value. To see how the division is actually implemented, the three ideal values should be replaced by the general slope/bias encoding scheme described in "Scaling" on page 3-5.

$$V_i = F_i 2^{E_i} Q_i + B_i$$

For the case where the slopes are one and the biases are zero for all signals, the solution of the resulting equation for the output stored integer, Q_a, is given below.

$$Q_a = 2^{E_b - E_c - E_a}(Q_b / Q_c)$$

This equation involves an integer division and some bit shifts. If $E_a \geq E_b - E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b - E_c$, then the bit shifts are to the left and the implementation can

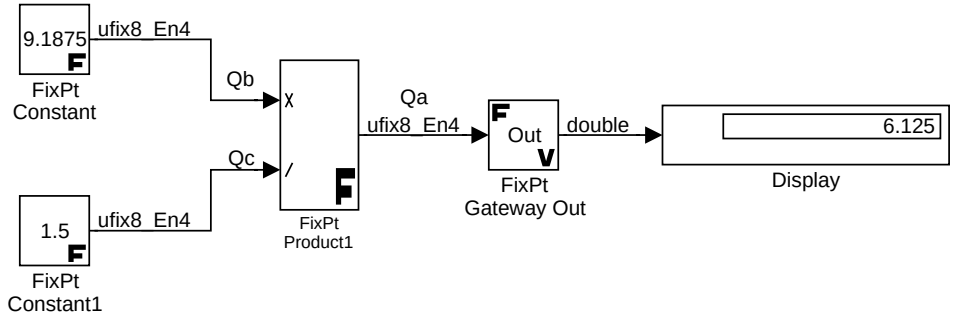
be more complicated. The essential issue is the output has more precision than the integer division provides. To get full precision, a *fractional* division is needed. The C programming language provides access to integer division only for fixed-point data types. Depending on the size of the numerator, some of the fractional bits may be obtained by performing a shift prior to the integer division. In the worst case, it may be necessary to resort to repeated subtractions in software.

In general, division of values is an operation that should be avoided in fixed-point embedded systems. Division where the output has more precision than the integer division (i.e., $E_a < E_b - E_c$) should be used with even greater reluctance. Division of signals with nonzero biases or mismatched slopes is not supported.

Example: The Division Process

Suppose you want to divide two numbers. Each of these numbers is represented by an 8-bit word, and each has a radix point-only scaling of 2^{-4}. Additionally, the output is restricted to an 8-bit word with radix point-only scaling of 2^{-4}.

The division of 9.1875 by 1.5000 is shown below.



For this example,

$$Q_a = 2^{-4-(-4)-(-4)}(Q_b/Q_c)$$

$$= 2^4(Q_b/Q_c)$$

Assuming a large data type was available, this could be implemented as

$$Q_a = \frac{(2^4 Q_b)}{Q_c}$$

where the numerator uses the larger date type. If a larger data type was not available, integer division combined with four repeated subtractions would be used. Both approaches produce the same result, with the former being more efficient.

Shifts

Nearly all microprocessors and digital signal processors support well-defined *bit-shift* (or simply *shift*) operations for integers. For example, consider the 8-bit unsigned integer 00110101. The results of a 2-bit shift to the left and a 2-bit shift to the right are shown below.

Shift Operation	Binary Value	Decimal Value
No shift (original number)	00110101	53
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

You can perform a shift with the Fixed-Point Blockset using either the Conversion block or the Gain block. The Conversion block shifts both the bits and radix point while the Gain block shifts the bits but not the radix point. These two modes of shifting as well as shifting to the right are discussed below.

Note Performing a “plain” or “raw” machine-level shift such as those given in the example above with the Fixed-Point Blockset is complicated by the available scaling options. Therefore, a single “Shift” block is not provided. For more information about scaling, refer to “Scaling” on page 3-5.

Shifting to the Right

Shifts to the right can be classified as a *logical* shift right or an *arithmetic* shift right. For a logical shift right, a 0 is incorporated into the most significant bit for each bit shift. For an arithmetic shift right, the most significant bit is recycled for each bit shift. With the Fixed-Point Blockset, shifting to the right follows these rules:

- For signed numbers, an arithmetic shift right is performed. Therefore, the most significant bit is recycled for each bit shift. For example, given the signed fixed-point number 10110.101, a bit shift two places to the right with the radix point unmoved yields the number 11101.101.
- For unsigned numbers, a logical shift right is performed. Therefore, the most significant bit is a 0 for each bit shift. For example, given the unsigned fixed-point number 10110.101, a bit shift two places to the right with the radix point unmoved yields the number 00101.101.

Shifting Bits and the Radix Point

With the Conversion block, you can perform a shift operation on the input by specifying the appropriate radix point-only scaling for the output. This block shifts both the bits and the radix point.

In most cases, you will perform a “plain” or “raw” shift. To perform such a shift using the Conversion block, you must configure the block’s dialog box this way:

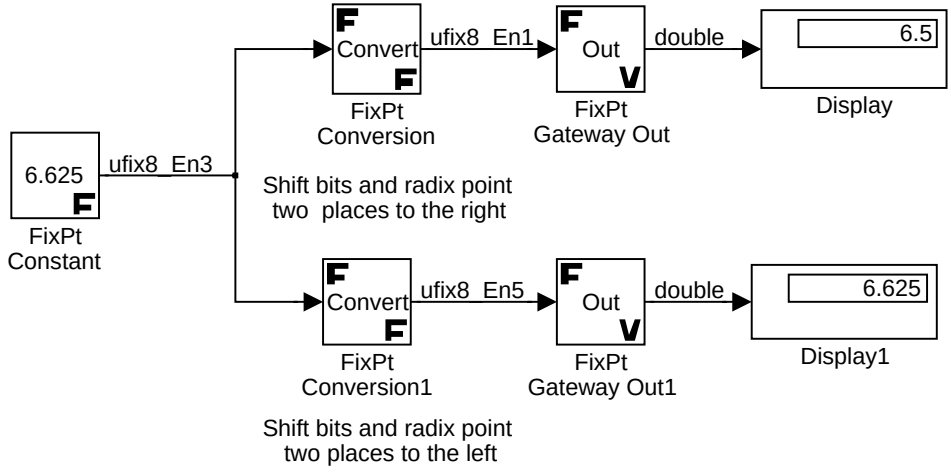
- The output data type is identical to the input data type.
- The rounding mode is set to `Floor`. Therefore, bits simply fall off the left or fall off the right when a shift occurs.
- Overflows wrap.
- The output scaling is specified to reflect the required shift.

For example, suppose you start with the fixed-point number 00110.101 (a decimal value of 6.625), which is characterized by the blockset as an 8-bit unsigned, generalized fixed-point number with radix point-only scaling of 2^{-3}. To shift the bits and radix point two places to the right, the input scaling of 2^{-3} is multiplied by 2^{2}, which yields a scaling of 2^{-1}. To shift the bits and radix point

two places to the left, the input scaling of 2^{-3} is multiplied by 2^{-2}, which yields as scaling of 2^{-5}. This situation is shown below.

Shift Operation	Scaling	Binary Value	Decimal Value
No shift (original number)	2^{-3}	00110.101	6.625
Shift right by 2 bits	2^{-1}	0000110.1	6.5
Shift left by 2 bits	2^{-5}	110.10100	6.625

The figure below shows the fixed-point model used to generate the above data.



Refer to Chapter 10, “Block Reference” for more information about the Conversion block.

Shifting Bits but Not the Radix Point

With the Gain block, you can perform a shift operation on the input by specifying the gain as a power of two. This block shifts only the bits and not the radix point.

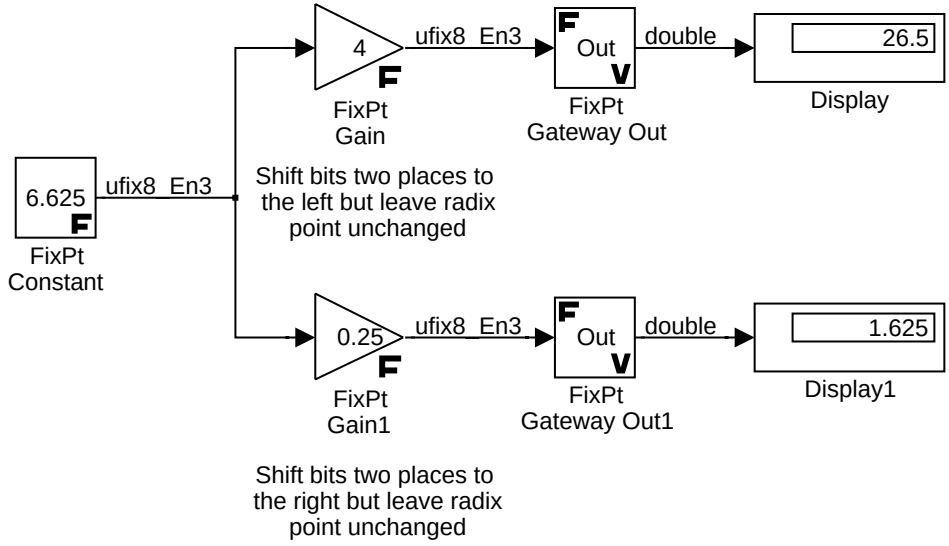
In most cases, you will perform a plain or raw shift. To perform such a shift using the Gain block, you must configure the block's dialog box this way:

- The output data type is identical to the input data type.
- The rounding mode is set to `Floor`. Therefore, bits simply fall off the left or fall off the right when a shift occurs.
- Overflows wrap.
- The gain is specified as the appropriate power of 2 to reflect the required shift.

For example, suppose you start with the same fixed-point number, 00110.101, defined above. To shift the bits two places to the left, a gain of 4 is specified, and to shift the bits two places to the right, a gain of 0.25 is specified. This situation is shown below.

Shift Operation	Gain Value	Binary Value	Decimal Value
N/A (original number)	2^{-3}	00110.101	6.625
Shift left by 2 bits	4	11010.100	26.5
Shift right by 2 bits	0.25	00001.101	1.625

The figure below shows the fixed-point model used to generate the above data.



Refer to Chapter 10, “Block Reference” for more information about the Gain block.

Example: Conversions and Arithmetic Operations

This example uses the FIR block to illustrate when parameters are converted from a double to a fixed-point number, when the input data type is converted to the output data type, and when the rules for addition and subtraction, and multiplication are applied. For details about conversions and operations, refer to "Parameter and Signal Conversions" on page 4-26 and "Rules for Arithmetic Operations" on page 4-30.

Note If a block can perform all four arithmetic operations, such as the FIR block, then the rules for multiplication and division are applied first.

Suppose you configure the FIR block for two outputs (SIMO mode) where the first output is given by

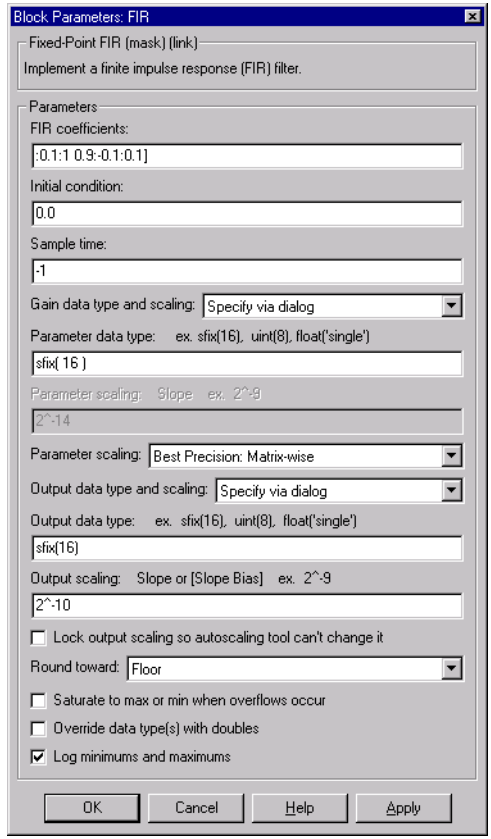
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2)$$

and the second output is given by

$$y_2(k) = 6 \cdot u(k) - 5 \cdot u(k-1)$$

Additionally, the initial values of $u(k-1)$ and $u(k-2)$ are given by 0.8 and 1.1, respectively and all inputs, parameters, and outputs have radix point-only scaling.

To configure the FIR block for this situation, you must specify the **FIR coefficient** parameter as `[13 11 -7; 6 -5 0]` and the **Initial condition** parameter as `[0.8 1.1]` as shown below in the dialog box below.



Parameter conversions and block operations are given below in the order in which they are carried out by the FIR block.

1 The **FIR coefficients** parameter is converted from doubles to the **Parameter data type** offline using round-to-nearest and saturation.

   The **Initial condition** parameter is converted from doubles to the input data type offline using round-to-nearest and saturation.

2 The coefficients and inputs are multiplied together for the initial time step for both outputs. For $y_1(0)$, the operations $13 \cdot u(0)$, $11 \cdot 0.8$, and $-7 \cdot 1.1$ are performed, while for $y_2(0)$, the operations $6 \cdot u(0)$ and $-5 \cdot 0.8$ are performed.

   The results of these operations are then converted to the **Output data type** using the specified rounding and overflow modes.

3 The sum is carried out for $y_1(0)$ and $y_2(0)$. Note that the rules for addition and subtraction are satisfied since the coefficients and inputs are already converted to the **Output data type**.

4 Steps **2** and **3** are repeated for subsequent time steps.

5

Realization Structures

Overview 5-2

Direct Form II 5-4

Series Cascade Form 5-7

Parallel Form 5-10

Overview

This chapter investigates how you can realize digital filters using the Fixed-Point Blockset.

The Fixed-Point Blockset addresses the needs of the control system and signal processing fields, and other fields where algorithms are implemented on fixed-point hardware. In signal processing, a digital filter is a computational algorithm that converts a sequence of input numbers to a sequence of output numbers. The algorithm is designed such that the output signal meets frequency-domain or time-domain constraints (desirable frequency components are passed, undesirable components are rejected). In general terms, a discrete transfer function controller is a form of a digital filter. However, a digital controller may contain nonlinear functions such as look-up tables in addition to a discrete transfer function. In this guide, the term "*digital filter*" is used when referring to discrete transfer functions.

The Fixed-Point Blockset does not attempt to standardize on one particular fixed-point digital filter design method. For example, a design can be done in continuous time and an "equivalent" discrete-time digital filter can be obtained using one of many transformation methods. Alternatively, digital filters can be directly designed in discrete time. After the digital filter is obtained, it can be realized for fixed-point hardware using any number of canonical forms. Typical canonical forms are the direct form, series form, and parallel form, all of which are outlined in this chapter.

For a given digital filter, the canonical forms describe a set of fundamental operations for the processor. Since there are an infinite number of ways to realize a given digital filter, the best realization must be made on a per-system basis. The canonical forms presented in this chapter optimize the implementation with respect to some factor, such as minimum number of delay elements. In general, when choosing a realization method, you must take these factors into consideration:

- **Cost**

  The cost of the realization might rely on minimal code and data size.

- **Timing constraints**

  Real-time systems must complete their compute cycle within a fixed amount of time. Some realizations might yield faster execution speed on different processors.

- **Output signal quality**

  The limited range and precision of the binary words used to represent real-world numbers will introduce errors. Some realizations are more sensitive to these errors than others.

The Fixed-Point Blockset allows you to evaluate various digital filter realization methods in a simulation environment. Following the development cycle outlined in “The Development Cycle” in Chapter 1, you can fine tune the realizations with the goal of reducing the cost (code and data size) or increasing signal quality. After the desired performance has been achieved, you can use the Real-Time Workshop to generate rapid prototyping C code and evaluate its performance with respect to your system’s real-time timing constraints. You can then modify the model based upon feedback from the rapid prototyping system.

The presentation of the various realization structures takes into account that a summing junction is a fundamental operator; thus you may find that the structures presented here look different from those in the fixed-point filter design literature. For each realization form, an example is provided using the transfer function shown below.

$$
\begin{aligned}
H_{ex}(z) &= \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}} \\
&= \frac{(1 + 0.5z^{-1})(1 + 1.7z^{-1} + z^{-2})}{(1 + 0.1z^{-1})(1 - 0.6z^{-1} + 0.9z^{-2})} \\
&= 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}
\end{aligned}
$$

Direct Form II

In general, a direct form realization refers to a structure where the coefficients of the transfer function appear directly as gain blocks. The direct form II realization method is presented as using the minimal number of delay elements, which is equal to *n*, the order of the transfer function denominator.

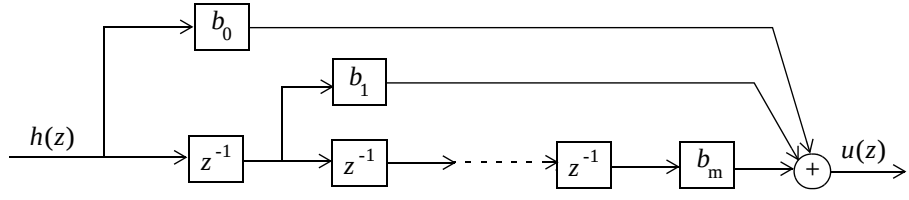
The canonical direct form II is presented as "Standard Programming" in *Discrete-Time Control Systems* by Ogata. It is known as the "Control Canonical Form" in *Digital Control of Dynamic Systems* by Franklin, Powell, and Workman.

You can derive the canonical direct form II realization by writing the discrete-time transfer function with input *e(z)* and output *u(z)* as

$$\frac{u(z)}{e(z)} = \frac{u(z)}{h(z)} \cdot \frac{h(z)}{e(z)}$$

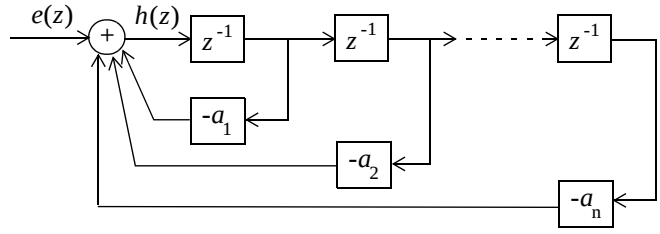
$$= \underbrace{(b_0 + b_1 z^{-1} + \ldots + b_m z^{-m})}_{\frac{u(z)}{h(z)}} \underbrace{\frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}}_{\frac{h(z)}{e(z)}}$$

The block diagram for *u(z)/h(z)* is shown below.



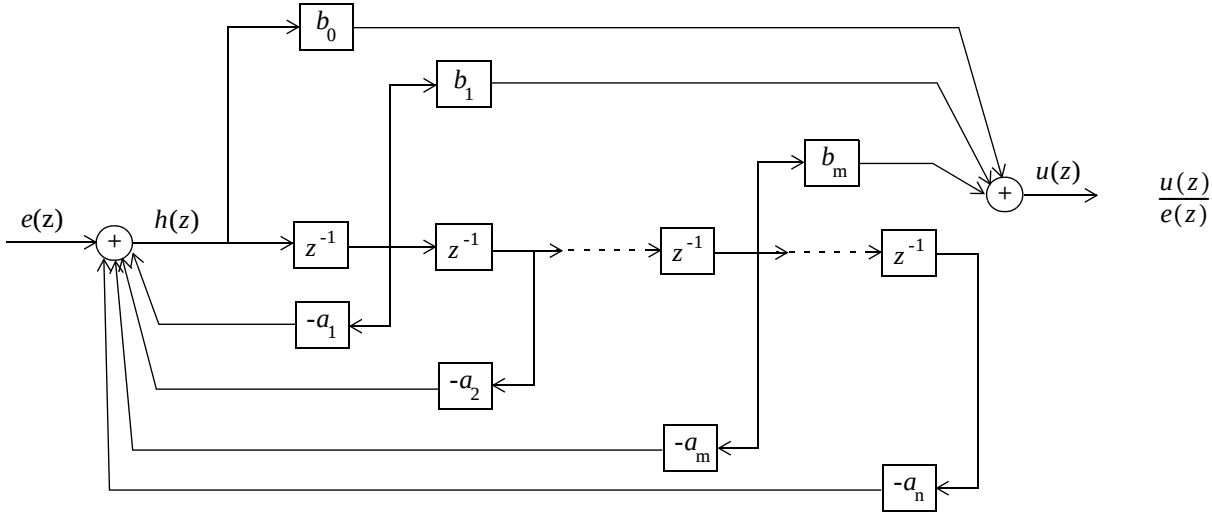
$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}$$

The block diagrams for $h(z)/e(z)$ is shown below.



$$\frac{h(z)}{e(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}$$

Combining these two block diagrams yields the direct form II diagram shown below. Notice that the feedforward part (top of block diagram) contains the numerator coefficients and the feedback part (bottom of block diagram) contains the denominator coefficients.



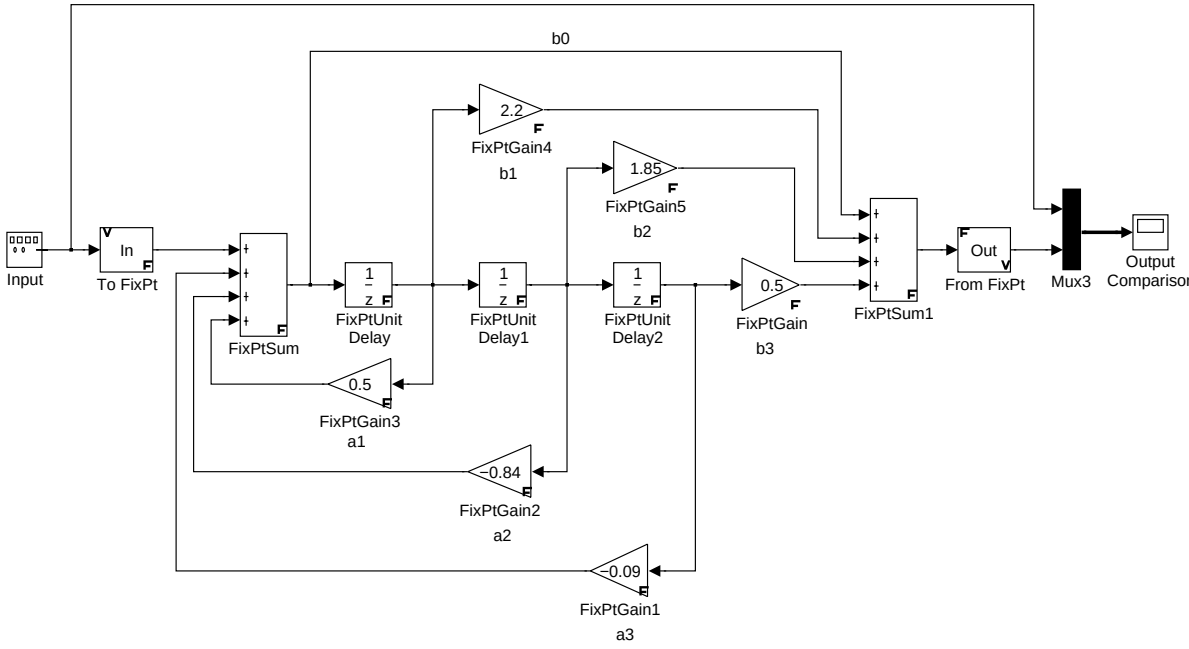
The direct form II example transfer function is given by

$$H_{ex}(z) = \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}}$$

The realization of $H_{ex}(z)$ using the Fixed-Point Blockset is shown below. You can display this model by typing

```
fxpdemo_direct_form2
```

at the MATLAB command line.

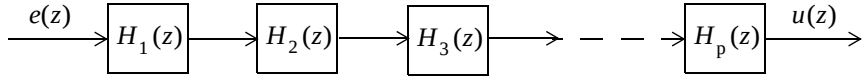


Series Cascade Form

In the canonical series cascade form, the transfer function *H(z)* is written as a product of first-order and second-order transfer functions.

$$H_i(z) = \frac{u(z)}{e(z)} = H_1(z) \cdot H_2(z) \cdot H_3(z) \ldots H_p(z)$$

This equation yields the canonical series cascade form shown below.



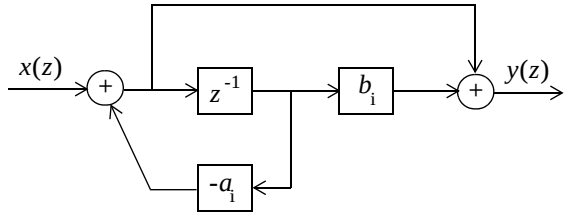
Factoring *H(z)* into $H_i(z)$ where $i = 1,2,3,\ldots,p$ can be done in a number of ways. Using the poles and zeros of *H(z)*, you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles and real zeros to produce either first-order or second-order transfer functions. You could also group two real zeros with a pair of conjugate complex poles or vice versa. Since there are many ways to obtain $H_i(z)$, it is desirable to compare the various groupings to see which produces the best results for the transfer function under consideration.

For example, one factorization of *H(z)* might be

$$\begin{aligned} H(z) &= H_1(z)H_2(z)\ldots H_p(z) \\ &= \prod_{i=1}^{j} \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^{p} \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}} \end{aligned}$$

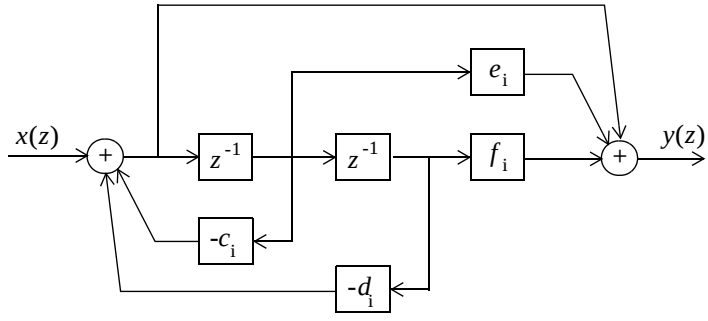
You must also take into consideration that the ordering of the individual $H_i(z)$'s will lead to systems with different numerical characteristics. You may want to try various orderings for a given set of $H_i(z)$'s to determine which gives the best numerical characteristics.

The first order diagram for *H(z)* is given below.



$$\frac{y(z)}{x(z)} = \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}}$$

The second order diagram for *H(z)* is given below.



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

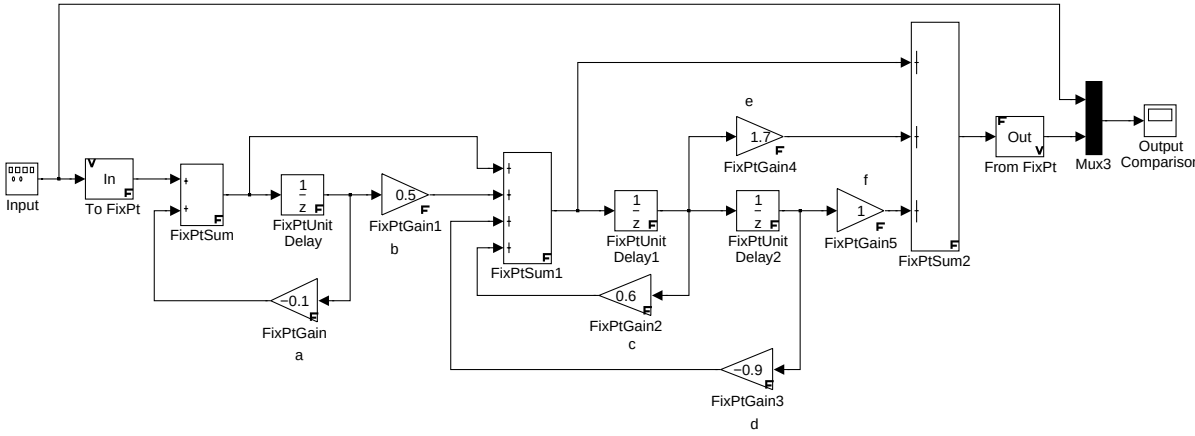
The series cascade form example transfer function is given by

$$H_{ex}(z) = \frac{(1 + 0.5z^{-1})(1 + 1.7z^{-1} + z^{-2})}{(1 + 0.1z^{-1})(1 - 0.6z^{-1} + 0.9z^{-2})}$$

The realization of $H_{ex}(z)$ using the Fixed-Point Blockset is shown below. You can display this model by typing

```
fxpdemo_series_cascade_form
```

at the MATLAB command line.

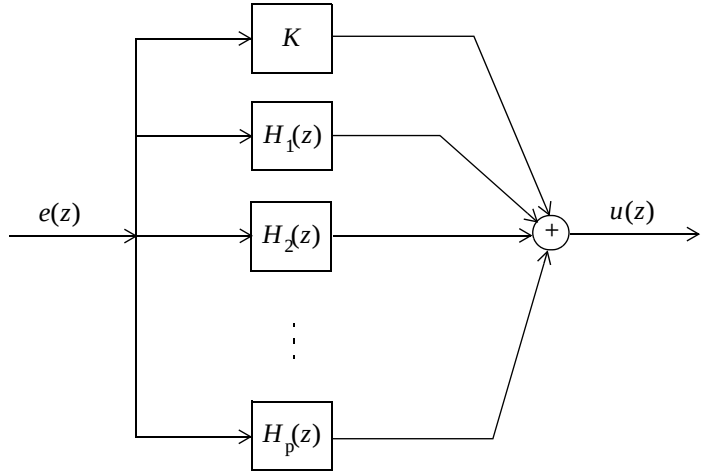


Parallel Form

In the canonical parallel form, the transfer function *H(z)* is expanded into partial fractions. *H(z)* is then realized as a sum of a constant, first-order, and second-order transfer functions as shown below.

$$H_i(z) = \frac{u(z)}{e(z)} = K + H_1(z) + H_2(z) + \ldots + H_p(z)$$

This expansion, where K is a constant and the $H_i(z)$ are the first and second-order transfer functions, is shown below.

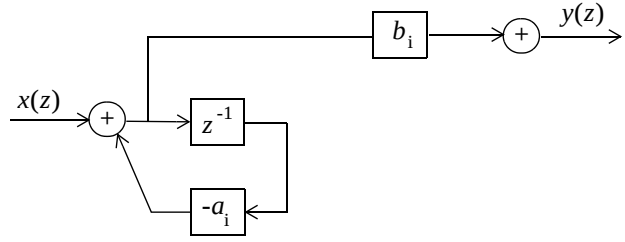


As in the series canonical form, there is no unique description for the first-order and second-order transfer function. Due to the nature of the Sum block, the ordering of the individual filters doesn't matter. However, because of the constant *K*, the first-order and second-order transfer functions can be chosen such that their forms are simpler than those for the series cascade form described in the preceding section. This is done by expanding *H(z)* as

$$H(z) = K + \sum_{i=1}^{j} H_i(z) + \sum_{i=j+1}^{p} H_i(z)$$

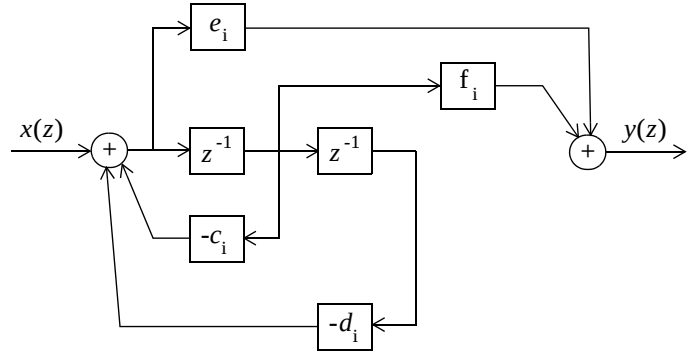
$$= K + \sum_{i=1}^{j} \frac{b_i}{1 + a_i z^{-1}} + \sum_{i=j+1}^{p} \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

The first order diagram for *H(z)* is shown below.



$$\frac{y(z)}{x(z)} = \frac{b_i}{1 + a_i z^{-1}}$$

The second order diagram for *H(z)* is shown below.



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

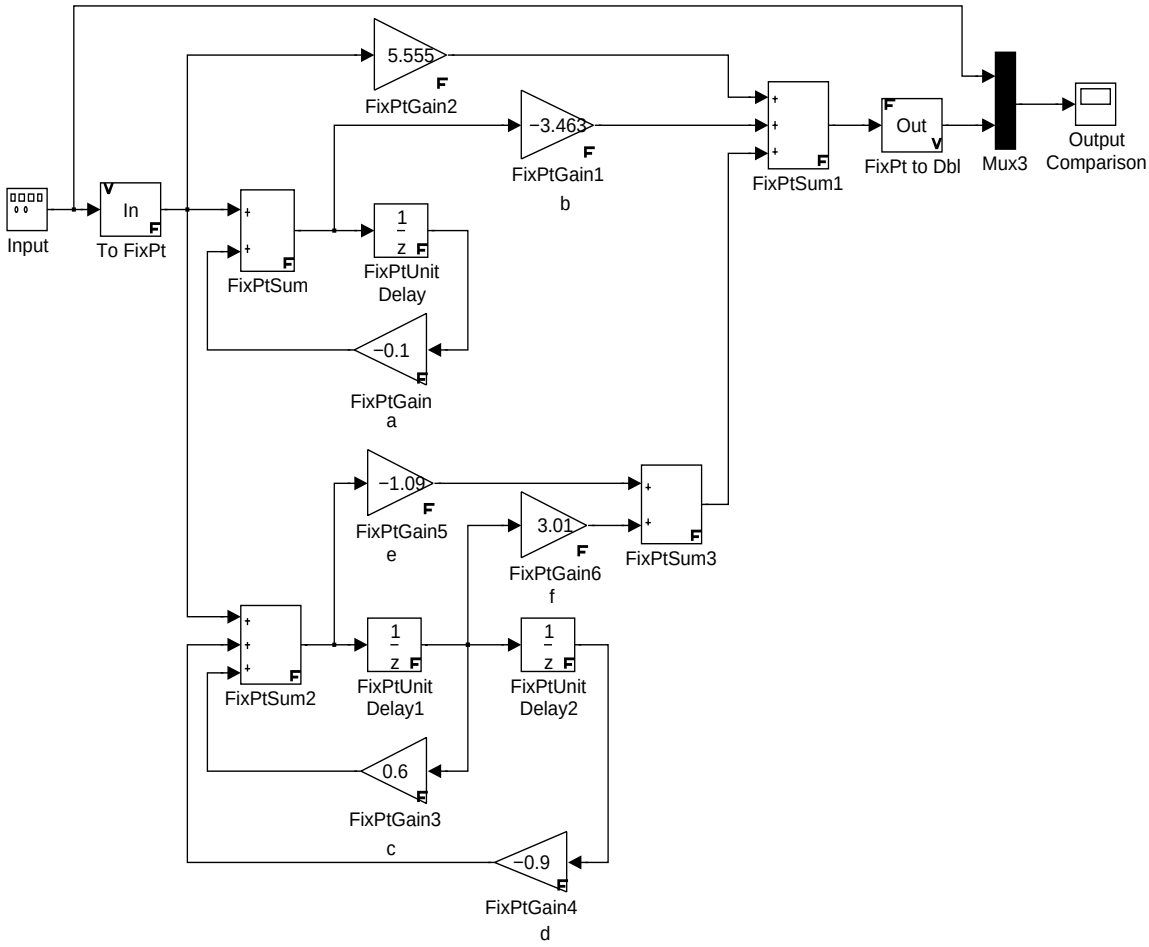
The parallel form example transfer function is given by

$$H_{ex}(z) = 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}$$

The realization of $H_{ex}(z)$ using the Fixed-Point Blockset is shown below. You can display this model by typing

```
fxpdemo_parallel_form
```

at the MATLAB command line.



6

Tutorial: Feedback Controller Simulation

Overview 6-2

Simulink Model of a Feedback Design 6-3

Idealized Feedback Design 6-6

Digital Controller Realization 6-7

Simulation Results 6-9
Simulation 1: Initial Guess at Scaling 6-9
Simulation 2: Global Override 6-12
Simulation 3: Automatic Scaling 6-15
Simulation 4: Individual Override 6-18

Overview

The purpose of this tutorial is to show you how to use fixed-point blocks to simulate a fixed-point feedback design using the Fixed-Point Blockset Interface tool. In doing so, many of the essential blockset features are demonstrated. These include:

- Output data type selection
- Output scaling
- Logging maximum and minimum simulation results
- The automatic scaling tool
- Overriding the output data type override for an entire model or an individual block

Simulink Model of a Feedback Design

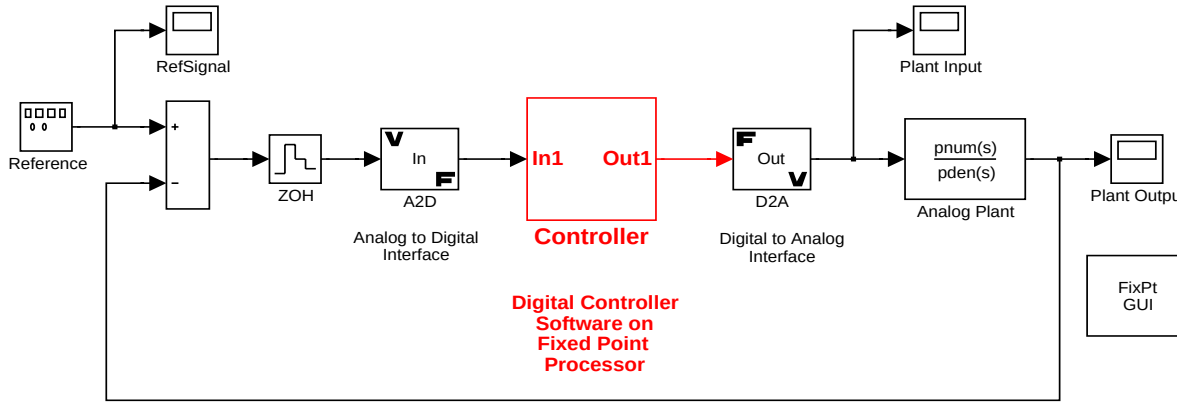
You can run the Simulink model of the feedback design by launching the MATLAB Demo browser and selecting the Scaling a Fixed-Point Control Design demo. You can launch the browser by typing

```
demo blockset 'Fixed Point'
```

at the command line, or by opening the Demos block found in the Fixed-Point Blockset library. Alternatively, you can access the model directly by typing its name at the command line.

```
fxpdemo_feedback
```

The MDL-file automatically runs the M-file `preload_feedback`, which populates the workspace with the required parameter values. The feedback design model is shown below.



The model consists of these blocks:

- **Reference**

  Simulink's Signal Generator block generates a continuous-time reference signal. It is configured to output a square wave.

- **ZOH**

  Simulink's Zero-Order Hold block samples and holds the continuous signal. This block is configured so that it quantizes the signal in time by an amount `tsamp` = 0.01 second.

- **Analog to Digital Interface**

  The analog to digital (A/D) interface consists of a Gateway In block that converts a Simulink double to a Fixed-Point Blockset data type. It represents any hardware that digitizes the amplitude of the analog input signal. In the real world, its characteristics are fixed.

- **Digital Controller**

  The digital controller is a subsystem that represents the software running on the hardware target. It is discussed in detail in "Digital Controller Realization" on page 6-7.

- **Digital to Analog Interface**

  The digital to analog (D/A) interface consists of a Gateway Out block that converts a Fixed-Point Blockset data type into a Simulink double. It represents any hardware that converts a digitized signal into an analog signal. In the real world, its characteristics are fixed.

- **Analog Plant**

  The analog plant is described by a transfer function, and is the object controlled by the digital controller. In the real world, its characteristics are fixed.

Simulation Setup

Setting up the fixed-point feedback controller simulation involves these steps:

1 Identify all design components

In the real world, there are design components with fixed characteristics (the hardware) and design components with characteristics that you can change (the software). The main components modeled in this feedback design are the A/D hardware, the digital controller, the D/A hardware, and the analog plant.

2 Develop a theoretical model of the plant and controller

For the feedback design used in this tutorial, the plant is characterized by a transfer function. The characteristics of the plant are unimportant for this tutorial, and are not discussed.

The digital controller model used in this tutorial is described by a *z*-domain transfer function and is implemented using a direct form realization.

3 Evaluate the behavior of the plant and controller

This is accomplished with a Bode plot. The evaluation is idealized since all numbers, operations, and states are double precision.

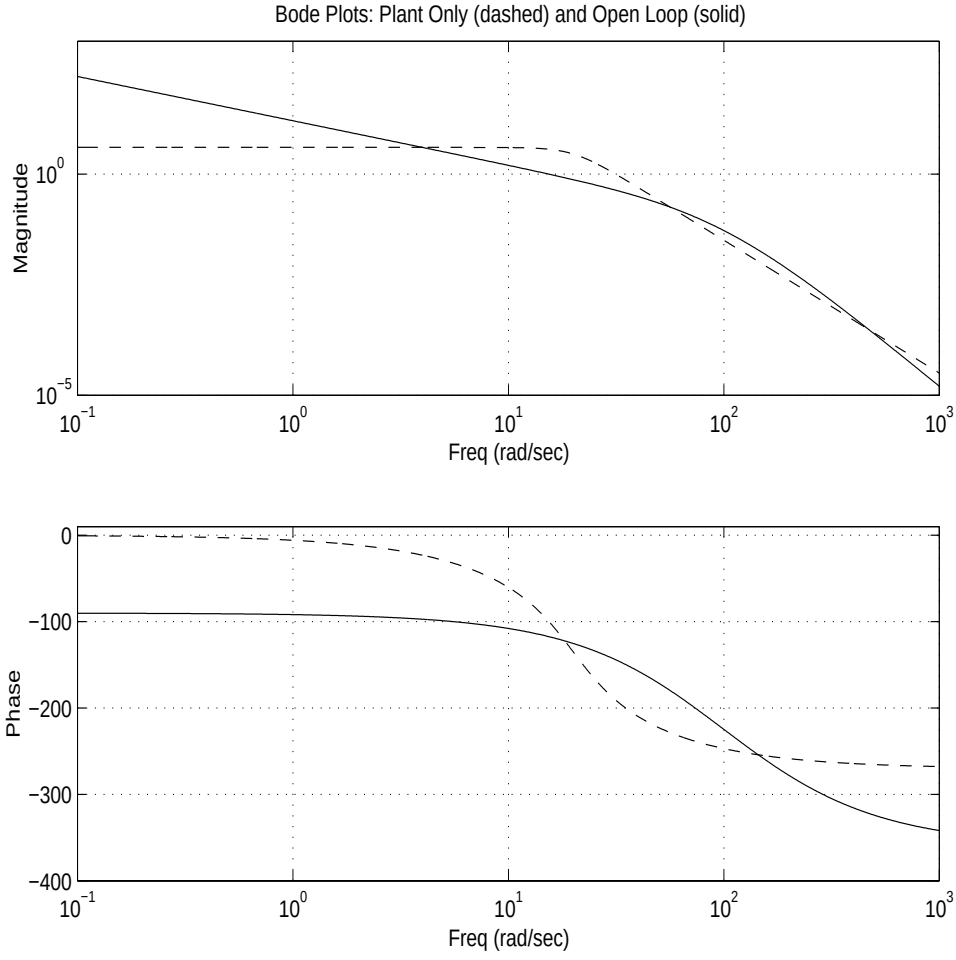
4 Simulate the system

The feedback controller design is simulated using Simulink and the Fixed-Point Blockset. Of course, in a simulation environment, you can treat all components (software *and* hardware) as though their characteristics are not fixed.

Idealized Feedback Design

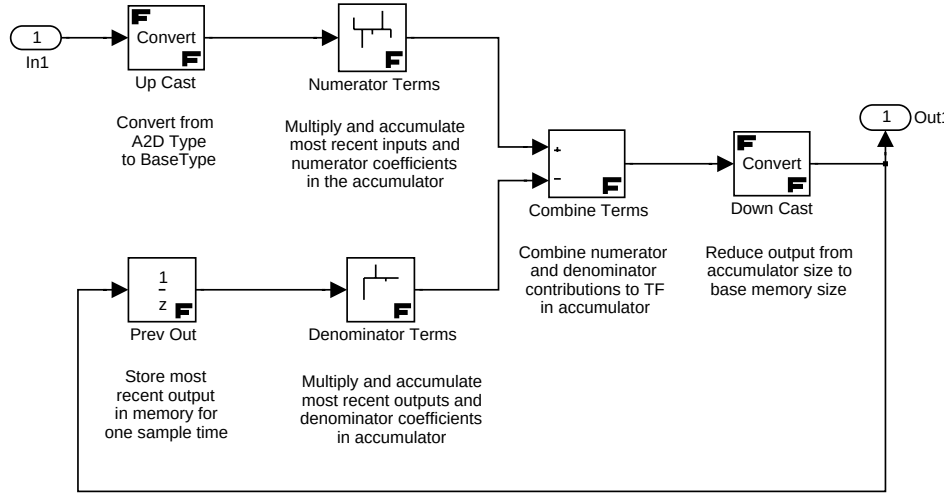
Open loop (controller and plant) and plant-only Bode plots for the Scaling a Fixed-Point Control Design demo are shown below. The open loop Bode plot results from a digital controller described in the idealized world of continuous time, and double precision coefficients, storage of states, and math operations.

The plant and controller design criteria are not important for the purposes of this tutorial. The Bode plots were created using the workspace variables produced by the `preload_feedback` M-file.



Digital Controller Realization

The digital controller is implemented using a fixed-point direct form realization. The target is a 16-bit processor. Variables and coefficients are generally represented using 16 bits, especially if these quantities are stored in ROM or global RAM. Use of 32-bit numbers is limited to temporary variables that exist briefly in CPU registers or in a stack. The realization is shown below.



The realization consists of these blocks:

- **Conversion**

  The Up Cast block connects the A/D hardware with the digital controller. It pads the output word of the A/D hardware with trailing zeros to a 16-bit number (the base data type). The Down Cast block represents taking the number from the CPU and storing it in RAM. The word size and precision are reduced to half that of the accumulator when converted back to the base data type.

- **FIR**

  These blocks represent a weighted sum carried out in the CPU target. The word size and precision used in the calculations reflect those of the accumulator. The Numerator Terms block multiplies and accumulates the

most recent inputs with the FIR numerator coefficients. The Denominator Terms block multiples and accumulates the most recent delayed inputs with the FIR denominator coefficients. The coefficients are stored in ROM using the base data type. The most recent inputs are stored in global RAM using the base data type.

- **Sum**

  The Combine Terms block represents the accumulator in the CPU. Its word size and precision are twice that of the RAM (double bits).

- **Unit Delay**

  The Prev Out block delays the feedback signal in memory by one sample period. The signals are stored in global RAM using the base data type.

Direct Form Realization

The controller directly implements this equation

$$y(k) = \sum_{i=0}^{N} b_i u(k-1) - \sum_{i=1}^{N} a_i y(k-1)$$

where:

- $u(k-1)$ represents the *input* from the previous time step.
- $y(k)$ represents the current output, and $y(k-1)$ represents the output from the previous time step.
- b_i represents the FIR numerator coefficients.
- a_i represents the FIR denominator coefficients.

The first summation in $y(k)$ represents multiplication and accumulation of the most recent inputs and numerator coefficients in the accumulator. The second summation in $y(k)$ represents multiplication and accumulation of the most recent inputs and denominator coefficients in the accumulator. Since the FIR coefficients, inputs, and outputs are all represented by 16-bit numbers (the base data type), any multiplication involving these numbers produces a 32-bit output (the accumulator data type).

Simulation Results

Using Simulink and the Fixed-Point Blockset, you can easily transition from a digital controller described in the ideal world of double precision numbers to one realized in the world of fixed-point numbers. The simulation approach used in this tutorial follows these steps:

1. Take an initial guess at the scaling. For this tutorial, an initial "proof of concept" simulation using a reasonable guess at the fixed-point word size and scaling is the first step in simulating the digital controller. This step is included only to illustrate how difficult it is to guess the best scaling.

2. Perform a global override of the fixed-point data types and scaling using double precision numbers. The maximum and minimum simulation values for each digital controller block are logged to the workspace.

3. Use the automatic scaling procedure. This procedure uses the doubles simulation values previously logged to the MATLAB workspace, and changes the scaling for each block that does not have its scaling fixed.

4. Perform a simulation on the "fixed" hardware block by overriding the data type with doubles. This simulation determines whether the A/D hardware warrants modification or replacement.

The feedback controller simulation is performed with the Fixed-Point Blockset Interface tool. You launch the Interface tool by selecting the GUI block within the `fxpdemo_feedback` model, by selecting **Fixed-Point** from the **Tools** menu in the model window, or by typing

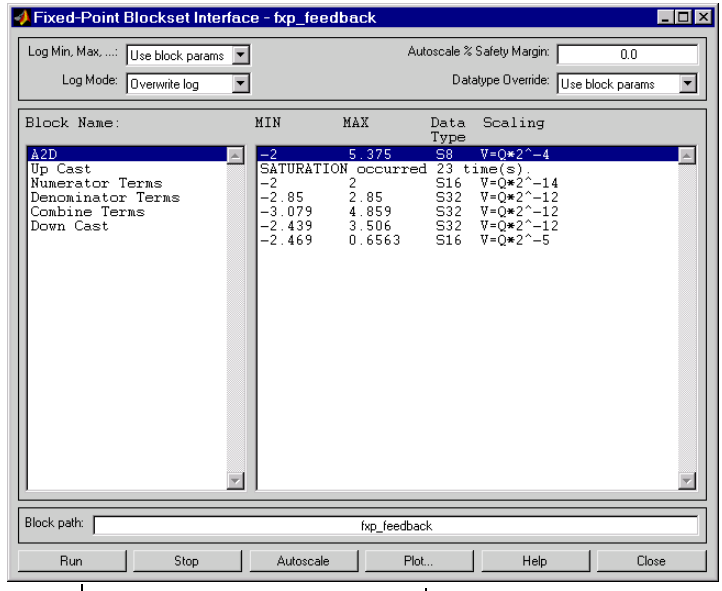
```
fxptdlg('fxpdemo_feedback')
```

at the command line. The four simulation trials are described in the following sections. The quality of the simulation results is determined by examining the input and output of the analog plant.

Simulation 1: Initial Guess at Scaling

The first simulation uses guesses for the scaling. In general, you won't need to perform this step, and this simulation is included to illustrate the difficulty of guessing at scaling.

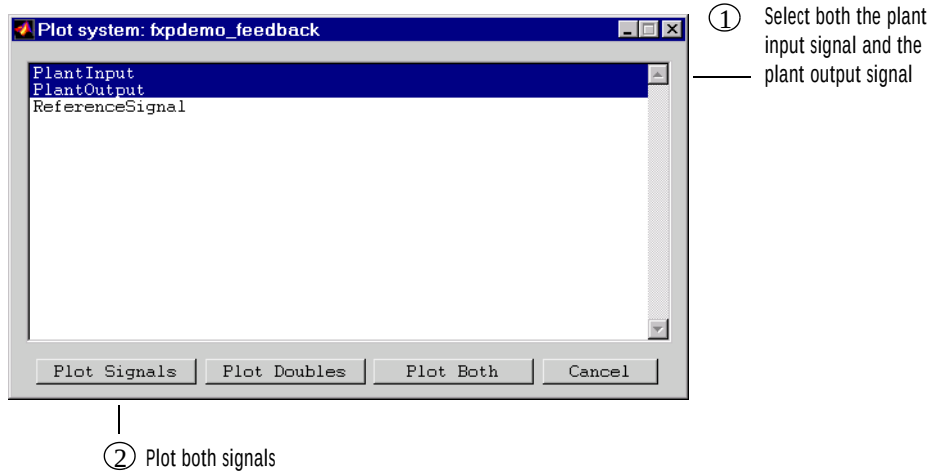
After you launch the Interface tool, press the **Run** button to run the simulation. When the simulation is finished, the interface displays the block name, the maximum and minimum simulation results, the data type, and the scaling for each block. You can then easily plot the results by pressing the **Plot** button, which launches the Plot System interface. The procedure for this simulation, and the simulation results are shown below.



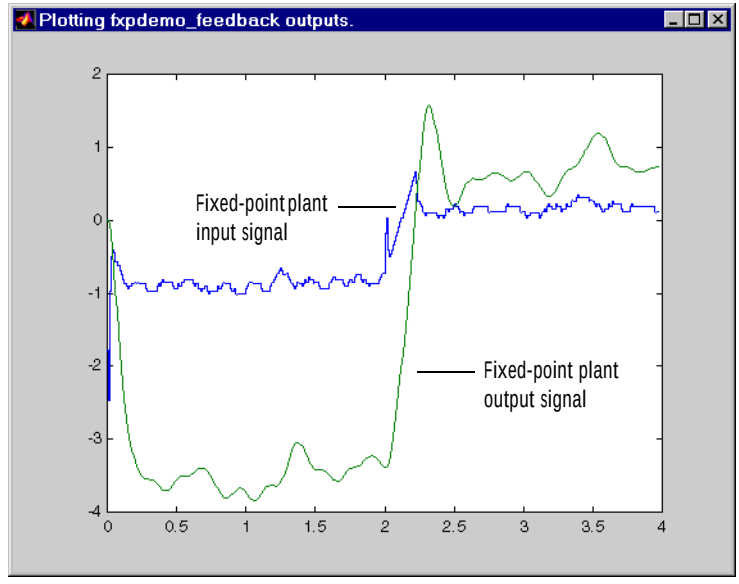
The display shows that the Up Cast block saturated 23 times, indicating a poor guess for the scaling. Refer to “Logging Simulation Results” on page 10-9 to learn about logging overflow information to the workspace.

The Plot System interface is shown below. This interface displays all MATLAB variable names that contain Scope block data for the current model. You configure the variable name with the Scope block’s **Properties** dialog box, which you launch by choosing the **Properties** toolbar button.

To plot the simulation results, select one or more variable names, and then select the appropriate plot button. This simulation plots the fixed-point signals for the plant input and the plant output.



The plant input signal and plant output signals are shown below. These signals reflect the initial guess at scaling.

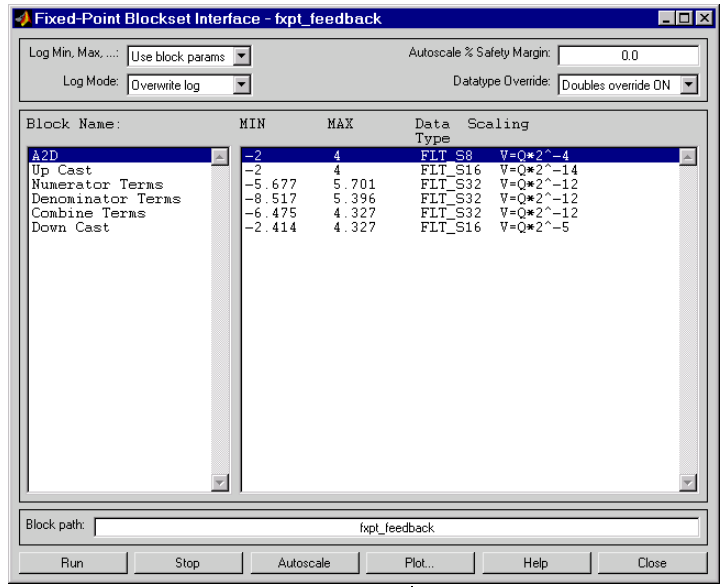


The Bode plot design sought to produce a well behaved linear response for the closed loop system. Clearly, the response is nonlinear. The nonlinear features are due to significant quantization effects. An important part of fixed-point design is finding scalings that reduce quantization effects to acceptable levels.

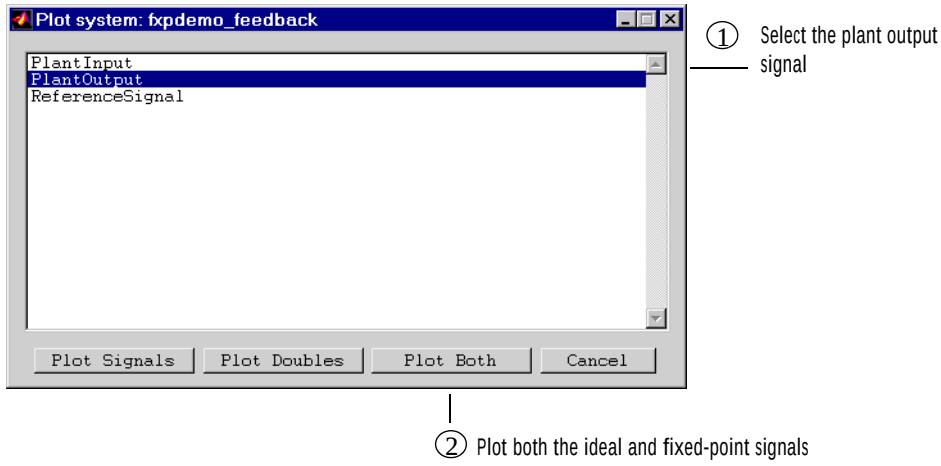
Simulation 2: Global Override

Prior to using the automatic scaling tool, a global override with doubles of the fixed-point data type is performed for every block. Using this feature, you can obtain ideal simulation limits. Additionally, you must log maximum and minimum simulation values for all blocks that are to be scaled. This is accomplished by checking the **Log minimums and maximums** check box for the relevant blocks, and accepting the **Log Min, Max** default value `Use block params`.

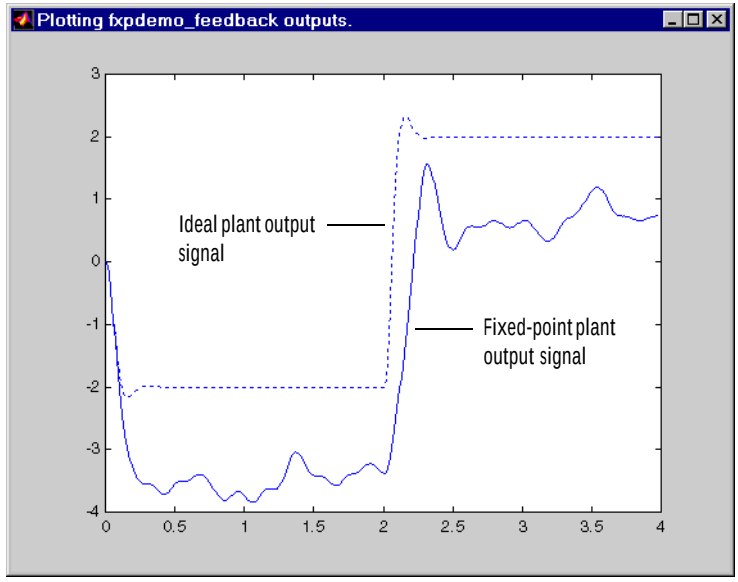
Global override with doubles is accomplished by configuring **Datatype Override** to `Doubles override ON`, and then running the simulation by selecting the **Run** button. The ideal and fixed-point plant output signals are then compared using the Plot System interface. The procedure for this simulation, and the simulation results are shown below.



The Plot System interface is shown below. This simulation plots both the fixed-point and ideal (double precision) signals for the plant output.



The ideal and fixed-point plant output signals are shown below. The ideal signal is produced by overriding the block output scaling.



Simulation 3: Automatic Scaling

Using the automatic scaling procedure, you can easily maximize the precision of the output data type while spanning the full simulation range. For a complex model, the absence of such a procedure can make achieving this goal tedious and time consuming.

Automatic scaling is performed for the Controller block. This block is a subsystem representing software running on the target, and requires optimization. Automatic scaling consists of these steps:

1. Configure **Autoscale % Safety Margin** to 20. This sets scaling so that the largest simulation value seen is at least 20% smaller than the maximum value allowed.

   The **Autoscale % Safety Margin** parameter value multiplies the "raw" simulation values by a factor of 1.2. Configuring this parameter to a value greater than 1 guarantees the simulation covers the largest possible range, although it does not necessarily mean the resolution improves. Since there is always some uncertainty when representing a real-world value with a fixed-point number with only a few simulations, using this parameter is recommended.

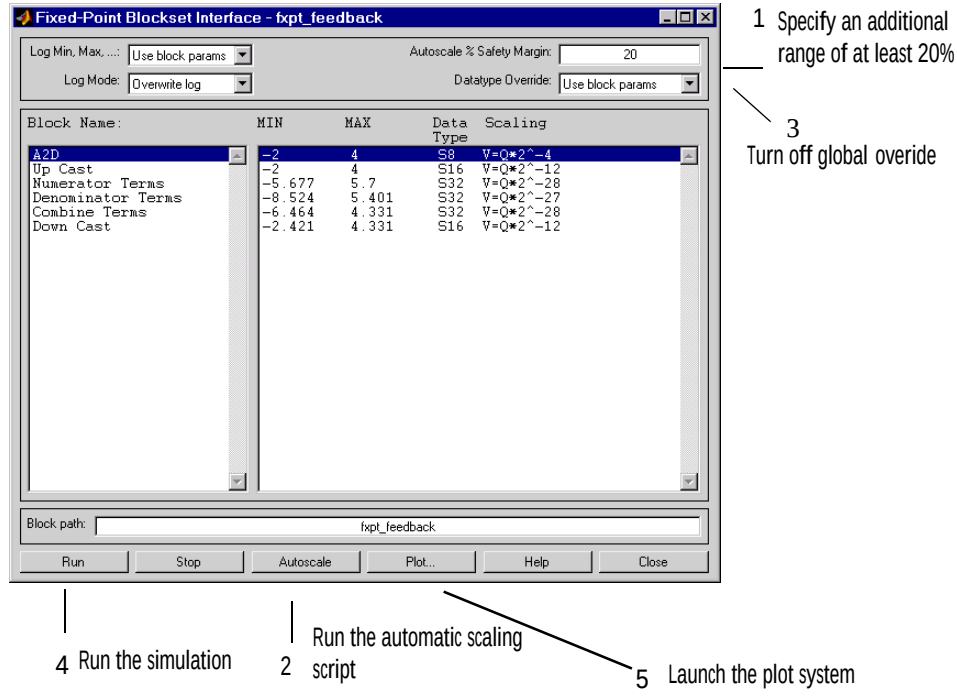
2. Run the `autofixexp` M-file script by selecting the **Autoscale** button. This script automatically changes the scaling on all fixed-point blocks that do not have their scaling locked, and that have their output data type specified as a generalized fixed-point number. It uses the minimum and maximum data logged from the previous simulation. The scaling changes such that the precision is maximized while the full range of simulation values are spanned for each block.

3. Turn off the global override with doubles by configuring **Datatype Override** to `Use block params`.

4. Run the simulation by selecting the **Run** button. The automatic scaling results generated by step 2 are captured from the MATLAB workspace and applied to the simulation.

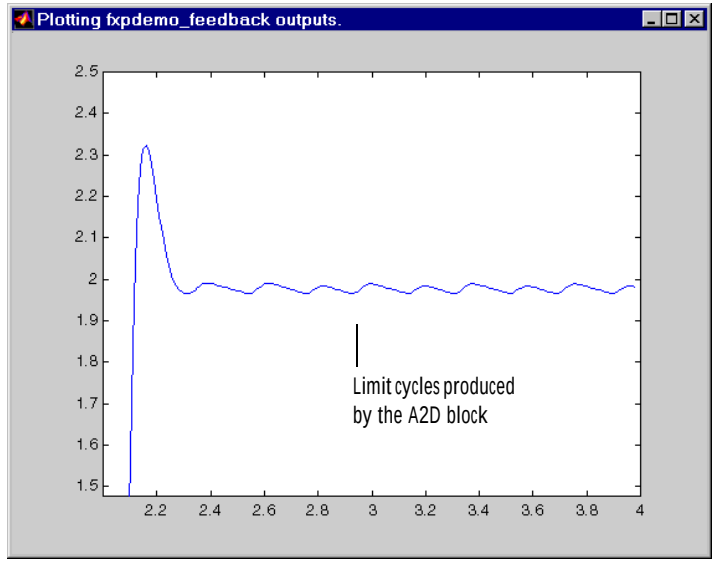
5. Launch the Plot System interface and plot the plant output signal.

The procedure and results for this simulation are shown below.

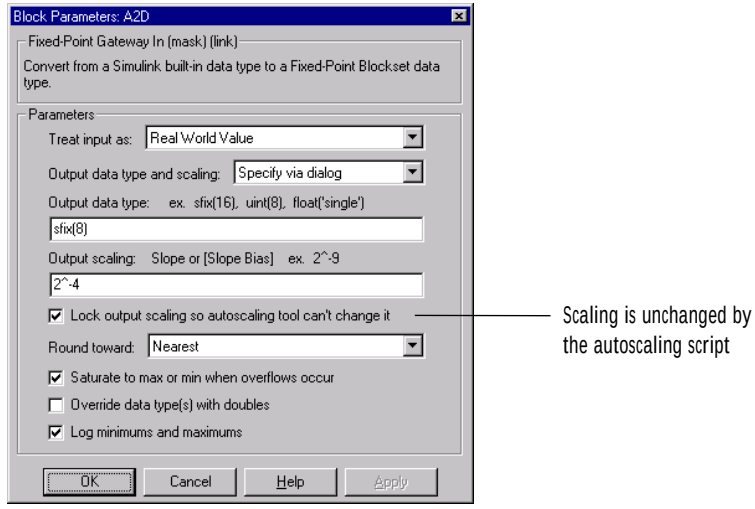


Each block is scaled based on its own maximum and minimum values obtained from the previous simulation using double-precision numbers. As shown above, the interface displays the new scaling for each block that had its scaling changed. This scaling is based on the raw simulation values multiplied by 1.2. Note that no saturations or overflows are reported.

A close-up of the plant output signal is shown below. Note that a steady-state has been achieved, but a small limit cycle is present in the steady state due to poor A/D design.

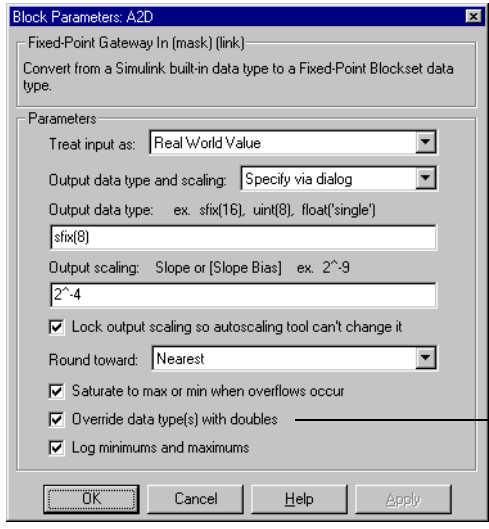


As shown below, the scaling of the A2D did not change because it was locked.



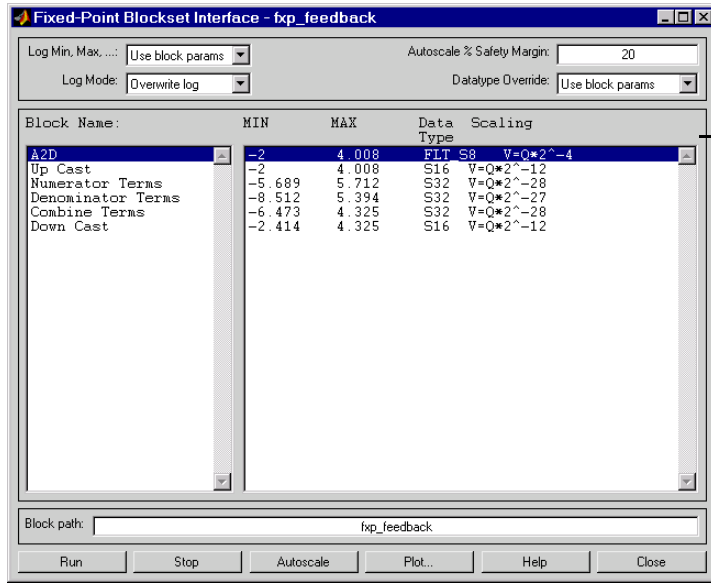
Simulation 4: Individual Override

The previous simulation optimized results for the fixed-point digital controller. In this simulation, the A2D block is configured so that it feeds the digital controller with doubles. This represents overriding the A/D hardware constraints, and is accomplished by checking the **Override with doubles** check box as shown below as shown below.



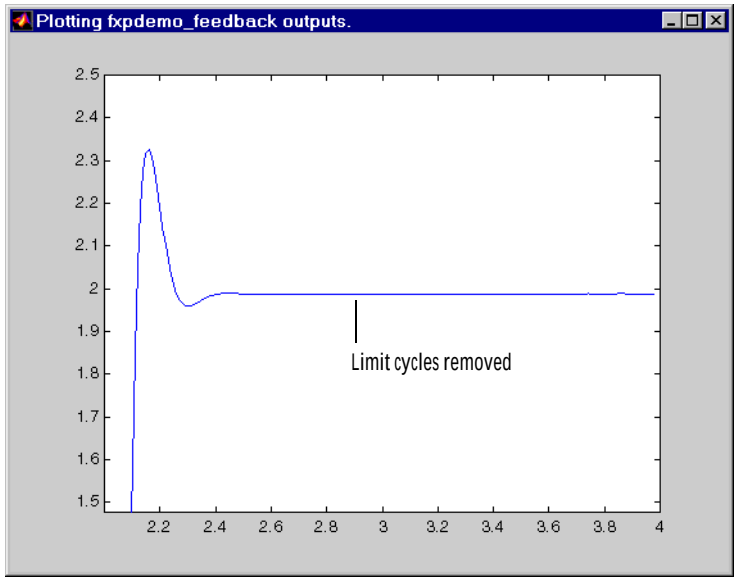
Note You can display the A2D dialog box, by double-clicking on the A2D entry in the interface.

The procedure and results for this simulation are shown below.



A close-up of the plant output signal is shown below. The limit cycle is no longer present in the steady state – confirmation of a poor A/D design. This means you

should replace the hardware, amplify the signal, or do some digital processing to better condition the signal.



7

Building Systems and Filters

Overview 7-2
Realizations and Data Types 7-2
Realizations and Scaling 7-2

Targeting an Embedded Processor 7-3
Size Assumptions 7-3
Operation Assumptions 7-3
Design Rules 7-4

Overview

The Fixed-Point Blockset provides several fixed-point filter and system realizations. These realizations are intended to be used as design templates so you can easily see how to build filters and systems that suit your particular application needs. Realizations are provided for state-space, integrator, derivative, and lead or lag systems. For more information about realization structures, refer to Chapter 5, “Realization Structures” or the references included in Appendix B.

Realizations and Data Types

In an ideal world where numbers, calculations, and storage of states have infinite precision and range, there are virtually an infinite number of realizations for the same system. In theory, these realizations are all identical to each other.

In the more realistic world of double-precision numbers, calculations, and storage of states, small nonlinearities are introduced due to the finite precision and range of floating-point data types. Therefore, each realization of a given system will produce different results. In most cases however, these differences are small.

In the world of fixed-point numbers where precision and range are limited, the differences in the realization results can be very large. Therefore, you must carefully select the data type, word size, and scaling for each realization element such that results are accurately represented. To assist you with this selection, design rules for modeling dynamic systems with fixed-point math are provided in “Targeting an Embedded Processor” on page 7-3.

Realizations and Scaling

As with all Fixed-Point Blockset models, you must select a scaling that gives the best precision, range, and performance for your specific fixed-point design.

The scaling for each filter and system demo is based on the default parameters. If these parameters are changed (for example, the magnitude of the input signal is increased), or if you are creating a new realization, you must define an appropriate scaling. For each system or filter, you can adjust the scaling manually with the dialog box, or automatically as illustrated in “Simulation Results” on page 6-9.

Targeting an Embedded Processor

This section describes issues that often arise when targeting a fixed-point design for use on an embedded processor. Rather than describe a specific microprocessor (micro) or digital signal processor (DSP), this section describes some general assumptions about integer sizes and operations available on embedded processors. These assumptions lead to design issues and design rules that may be useful for your specific fixed-point design.

Size Assumptions

Embedded processors are typically characterized by a particular bit size. For example, the terms “8-bit micro,” “32-bit micro,” or “16-bit DSP” are common. It is generally safe to assume that the processor is predominantly geared to processing integers of the specified bit size. Integers of the specified bit size are referred to as the *base data type*. Additionally, the processor typically provides some support for integers that are twice as wide as the base data type. Integers consisting of double bits are referred to as the *accumulator data type*. For example a 16-bit micro has a 16-bit base data type and a 32-bit accumulator data type.

Although other data types may be supported by the embedded processor, this section describes only the base and accumulator data types.

Operation Assumptions

The embedded processor operations discussed in this section are limited to the needs of a basic simulation diagram. Basic simulations use multiplication, addition, subtraction, and delays. Fixed-point models also need shifts to do scaling conversions. For all these operations, the embedded processor should have native instructions that allow the base data type as inputs. For accumulator-type inputs, the processor typically supports addition, subtraction, and delay (storage/retrieval from memory), but not multiplication.

Multiplication is typically not supported for accumulator-type inputs due to complexity and size issues. A difficulty with multiplication is that the output needs to be twice as big as the inputs for full precision. For example, multiplying two 16-bit numbers requires a 32-bit output for full precision. The need to handle the outputs from a multiply operation is one of the reasons embedded processors include accumulator-type support. However, if multiplication of accumulator-type inputs is also supported, then there is a

need to support a data type that is twice as big as the accumulator type. To restrict this additional complexity, multiplication is typically not supported for inputs of the accumulator type.

Design Rules

The important design rules that you should be aware of when modeling dynamic systems with fixed-point math are given below.

Design Rule 1: Only Multiply Base Data Types

It is best to multiply only inputs of the base data type. Embedded processors typically provide an instruction for the multiplication of base-type inputs but not for the multiplication of accumulator-type inputs. If necessary, a multiplication of accumulator-type inputs could be handled by combining several instructions. However, this can lead to large, slow embedded code.

Blocks to convert inputs from the accumulator-type to the base-type can be inserted prior to multiply or gain blocks if needed.

Design Rule 2: Delays Should Use the Base Data Type

There are two general reasons why a unit delay should use only base-type numbers. First, the unit delay essentially stores a variable's value to RAM, and one time step later, retrieves that value from RAM. Because the value must be in memory from one time step to the next, the RAM must be exclusively dedicated to the variable and can't be shared or used for another purpose. Using accumulator-type numbers instead of the base data type doubles the RAM requirements, which can significantly increase the cost of the embedded system. The second reason is that the unit delay typically feeds into a gain block. The multiplication design rule requires that the input (the unit delay signal) use the base data type.

Design Rule 3: Temporary Variables Can Use the Accumulator Data Type

Except for unit delay signals, most signals are not needed from one time step to the next. This means that the signal values can be temporarily stored in memory that is shared and reused. This shared and reused memory can be RAM or it can simply be registers in the CPU. In either case, storing the value as an accumulator data type is not much more costly than storing it as a base data type.

Design Rule 4: Summation Can Use the Accumulator Data Type

Addition and subtraction can use the accumulator data type if there is justification. The typical justification is reducing the buildup of errors due to round-off or overflow. For example, a common filter operation is a weighted sum of several variables. Multiplying a variable by a weight will naturally produce a product of the accumulator type. Before summing, each product could be converted back to the base data type. This approach introduces round-off error into each part of the sum. Alternatively, the products can be summed using the accumulator data type, and the final sum can be converted to the base data type. Round-off error is introduced in just one point and the precision will generally be better. The cost of doing an addition or subtraction using accumulator-type numbers is slightly more expensive, but if there is justification, it is usually worth the cost.

8

Producing Lookup Table Data

Overview 8-2

Worst Case Error for a Lookup Table 8-3
Example: Square Root Function 8-3

Creating Lookup Tables for a Sine Function 8-5
Parameters for fixpt_look1_func_approx 8-5
Setting Function Parameters for the Lookup Table 8-6
Example 1: Using errmax with Unrestricted Spacing 8-7
Example 2: Using nptsmax with Unrestricted Spacing . . . 8-9
Restricting the Spacing 8-10
Example 3: Using errmax with Even Spacing 8-11
Example 4: Using nptsmax with Even Spacing 8-12
Example 5: Using errmax with Power of Two Spacing 8-13
Example 6: Using nptsmax with Power of Two Spacing . . . 8-15
Specifying Both errmax and nptsmax 8-16
Comparing the Examples 8-17

Summary of the Lookup Table Functions 8-19

Effect of Spacing on Speed, Error, and Memory Usage . 8-20
Data ROM Required 8-21
Determining Out-of-Range Inputs 8-22
Determining the Location of the Input 8-22
Performing Interpolation. 8-24
Conclusion 8-26

Overview

A function lookup table is a method of approximating a function by a table with a finite number of points (X,Y). Function lookup tables are essential to many fixed-point applications. The function you want to approximate is called the *ideal function*. The X values of the lookup table are called the *breakpoints*. You approximate the value of the ideal function at a point by linearly interpolating between the two adjacent breakpoints closest to the point.

In creating the points for a function lookup table, you generally want to achieve one or both of the following goals:

- Minimize the worst case error for a specified maximum number of breakpoints
- Minimize the number of breakpoints for a specified maximum allowed error

This tutorial shows you how to create function lookup tables using the function `fixpt_look1_func_approx`. You can optimize the lookup table to minimize the number of data points, the error, or both. You can also restrict the spacing of the breakpoints to be even or even powers of two, in order to speed up computations using the table.

This tutorial also explains how to use the function `fixpt_look1_func_plot` to find the worst case error of a lookup table and plot the errors at all points.

The tutorial covers the following topics:

- "Worst Case Error for a Lookup Table" on page 8-3 explains how to use the function `fixpt_look1_func_plot` to find the worst case error and plot the fixed point approximation and the errors.
- "Creating Lookup Tables for a Sine Function" on page 8-5 presents several examples that illustrate how to use the function `fixpt_look1_func_approx` to create lookup tables.
- "Summary of the Lookup Table Functions" on page 8-19
- "Effect of Spacing on Speed, Error, and Memory Usage" on page 8-20 explains how restricting the spacing affects computational speed and memory usage.

Worst Case Error for a Lookup Table

This section explains the worst case error of a lookup table, and how to find the worst case error using the function `fixpt_look1_func_plot`. It gives a simple example of the worst case error of a lookup table for the square root function.

The error at any point of a function lookup table is the absolute value of the difference between the ideal function at the point and the corresponding Y value found by linearly interpolating between the adjacent breakpoints. The *worst case error*, or *maximum absolute error*, of a lookup table is the maximum absolute value of all errors in the interval containing the breakpoints.

For example, if the ideal function is the square root, and the breakpoints of the lookup table are 0, .25 and 1, then in a perfect implementation of the lookup table, the worst case error is 1/8 = .125, which occurs at the point 1/16 = .0625. In practice, the error could be greater, depending on the fixed point quantization and other factors.

Example: Square Root Function

This example shows how to use the function `fixpt_look1_func_plot` to find the maximum absolute error for the simple lookup table whose breakpoints are 0, .25, and 1. The corresponding Y data points of the lookup table, which you find by taking the square roots of the breakpoints, are 0, .5 and 1.

To use the function `fixpt_look1_func_plot`, you need to first define its parameters. To do so, type the following at the MATLAB prompt.

```
funcstr='sqrt(x)'; %Define the square root function
xdata=[0;.25;1]; %Set the breakpoints
ydata=sqrt(xdata); %Find the square root of the breakpoints
xmin = 0; %Set the minimum breakpoint
xmax = 1; %Set the maximum breakpoint
xdt = ufix(16); %Set the x data type
xscale = 2^-16; %Set the x data scaling
ydt = sfix(16); %Set the y data type
yscale = 2^-14; %Set the y data scaling
rndmeth = 'Floor'; %Set the rounding method
```

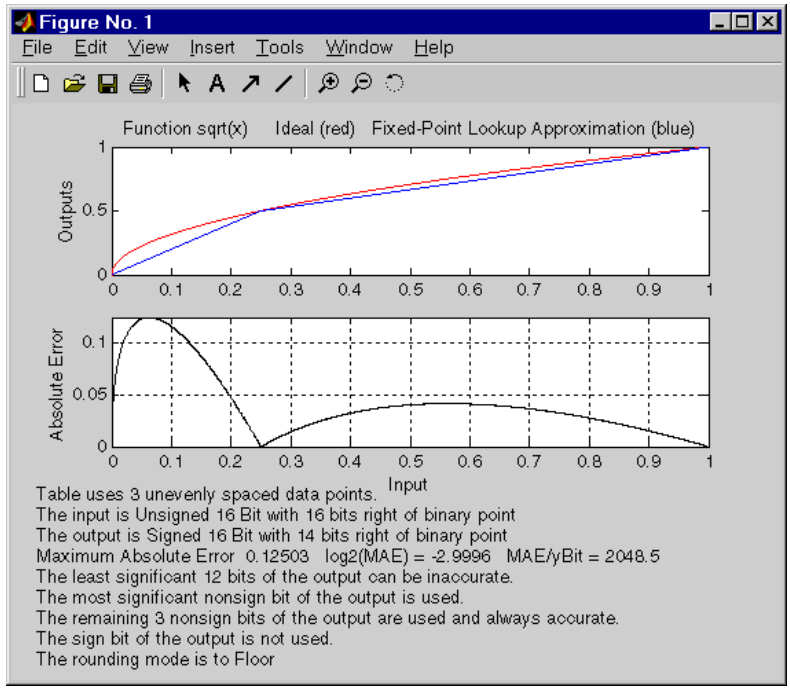
Next, type

```
errworst=fixpt_look1_func_plot(xdata,ydata,funcstr,...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

This returns the worst case error of the lookup table as the variable `errworst`.

```
errworst =
    0.1250
```

It also generates the plots shown below.



The upper box (Outputs) displays a plot of the square root function, and a plot of the fixed-point lookup approximation underneath. The approximation is found by linear interpolating between the breakpoints. The lower box (Absolute Error) displays the errors at all points in the interval from 0 to 1. Notice that the maximum absolute error occurs at .0625. The error at the breakpoints is 0.

Creating Lookup Tables for a Sine Function

This section explains how to use the function `fixpt_look1_func_approx` to create lookup tables. It gives examples that show how to create lookup tables for the function $\sin(2\pi x)$ on the interval from 0 to .25. The section covers:

- "Parameters for fixpt_look1_func_approx" on page 8-5
- "Setting Function Parameters for the Lookup Table" on page 8-6
- "Example 1: Using errmax with Unrestricted Spacing" on page 8-7
- "Example 2: Using nptsmax with Unrestricted Spacing" on page 8-9
- "Restricting the Spacing" on page 8-10
- "Example 3: Using errmax with Even Spacing" on page 8-11
- "Example 4: Using nptsmax with Even Spacing" on page 8-12
- "Example 5: Using errmax with Power of Two Spacing" on page 8-13
- "Example 6: Using nptsmax with Power of Two Spacing" on page 8-15
- "Specifying Both errmax and nptsmax" on page 8-16
- "Comparing the Examples" on page 8-17

Parameters for fixpt_look1_func_approx

To use the function `fixpt_look1_func_approx`, you must first define its parameters. The required parameters for the function are:

- `funcstr` – The ideal function
- `xmin` – The minimum input of interest
- `xmax` – The maximum input of interest
- `xdt` – The x data type
- `xscale` – The x data scaling
- `ydt` – The y data type
- `yscale` – The y data scaling
- `rndmeth` – The rounding method

In addition there are three optional parameters:

- `errmax` – The maximum allowed error of the lookup table
- `nptsmax` – The maximum number of points of the lookup table

- spacing – The allowed spacing between breakpoints

You must use at least one of the parameters errmax and nptsmax. Typical settings for these parameters are given in the next section "Setting Function Parameters for the Lookup Table" on page 8-6.

Using Only errmax

If you use only the errmax parameter, without nptsmax, the function creates a lookup table with the fewest points, for which the worst case error is at most errmax. See "Example 1: Using errmax with Unrestricted Spacing" on page 8-7.

Using Only nptsmax

If you use only the nptsmax parameter, without errmax, the function creates a lookup table with at most nptsmax points, which has the smallest worse case error. See "Example 2: Using nptsmax with Unrestricted Spacing" on page 8-9.

The section "Specifying Both errmax and nptsmax" on page 8-16 describes how the function behaves when you specify both errmax and nptsmax.

Spacing

You can use the optional spacing parameter to restrict the spacing between breakpoints of the lookup table. The options are:

- 'unrestricted' – The default.
- 'even' – The distance between any two adjacent breakpoints is the same.
- 'pow2' – The distance between any two adjacent breakpoints is the same and the distance is a power of two.

The section "Restricting the Spacing" on page 8-10 and the examples that follow it explain how to use the spacing parameter.

Setting Function Parameters for the Lookup Table

To do the examples in this section, you must first set parameter values for the fixpt_look1_func_approx function. To do so, type the following at the MATLAB prompt.

```
funcstr = 'sin(2*pi*x)'; %Define the sine function
xmin = 0; %Set the minimum input of interest
xmax = 0.25; %Set the maximum input of interest
```

```
xdt = ufix(16); %Set the x data type
xscale = 2^-16; %Set the x data scaling
ydt = sfix(16); %Set the y data type
yscale = 2^-14; %Set the y data scaling
rndmeth = 'Floor'; %Set the rounding method
errmax = 2^-10; %Set the maximum allowed error
nptsmax = 21; %Specify the maximum number of points
```

If you exit MATLAB after typing these commands, you must retype them before trying any of the other examples in this section.

Example 1: Using errmax with Unrestricted Spacing

The first example shows how to create a lookup table that has the fewest data points for a specified worst case error, with unrestricted spacing. Before trying the example, enter the same parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-6, if you have not already done so in this MATLAB session.

You specify the maximum allowed error by typing

```
errmax = 2^-10;
```

Creating the Lookup Table

To create the lookup table, type

```
[xdata,ydata,errworst]=fixpt_look1_func_approx(funcstr,...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax);
```

Note that the `nptsmax` and `spacing` parameters are not specified.

The function returns three variables:

- `xdata`, the vector of breakpoints of the lookup table
- `ydata`, the vector found by applying ideal function, $\sin(2\pi x)$, to `xdata`
- `errworst`, which specifies the maximum possible error in the lookup table.

The value of `errworst` is less than or equal to the value of `errmax`.

You can find the number of X data points by typing

```
length(xdata)
```

```
ans =

    16
```

This means that 16 points are required to approximate sin(2π*x)* to within the tolerance specified by `errmax`.

You can display the maximum error by typing `errworst`. This returns

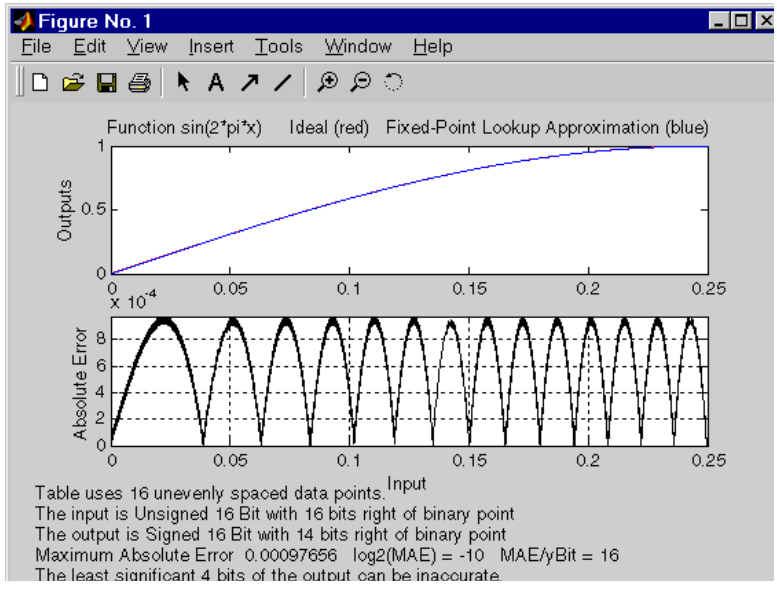
```
errworst =
  9.7656e-004
```

Plotting the Results

You can plot the output of the function `fixpt_look1_func_plot` by typing

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth);
```

The resulting plots are shown below.



The upper plot shows the ideal function, sin(2πx) and the fixed-point lookup approximation between the breakpoints. In this example, the ideal function

and the approximation are so close together that the two graphs appear to coincide. The lower plot displays the errors.

In this example, the Y data points, returned by the function `fixpt_look1_func_approx` as `ydata`, are equal to the ideal function applied to the points in `xdata`. However, you can define a different set of values for `ydata` after running `fixpt_look1_func_plot`. This can sometimes reduce the maximum error.

You can also change the values of `xmin` and `xmax` in order to evaluate the lookup table on a subset of the original interval.

To find the new maximum error after changing `ydata`, `xmin` or `xmax`, type

```
errworst=fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,...
xdt,xscale,ydt,yscale,rndmeth)
```

Example 2: Using nptsmax with Unrestricted Spacing

The next example shows how to create a lookup table that minimizes the worst case error for a specified maximum number of data points, with unrestricted spacing. Before starting the example, enter the same parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 8-6, if you have not already done so in this MATLAB session.

Setting the Number of Breakpoints

You specify the number of breakpoints in the lookup table by typing

```
nptsmax = 21;
```

Creating the Lookup Table

Next, type

```
[xdata,ydata,errworst]= fixpt_look1_func_approx(funcstr,
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax);
```

The empty brackets, `[]`, tell the function to ignore the parameter `errmax`, which is not used in this example. Omitting `errmax` causes the function `fixpt_look1_func_approx` to return a lookup table of size specified by `nptsmax`, with the smallest worst case error.

The function returns a vector `xdata`, with 21 points. You can find the maximum error for this set of points is given by typing `errworst` at the MATLAB prompt. This returns

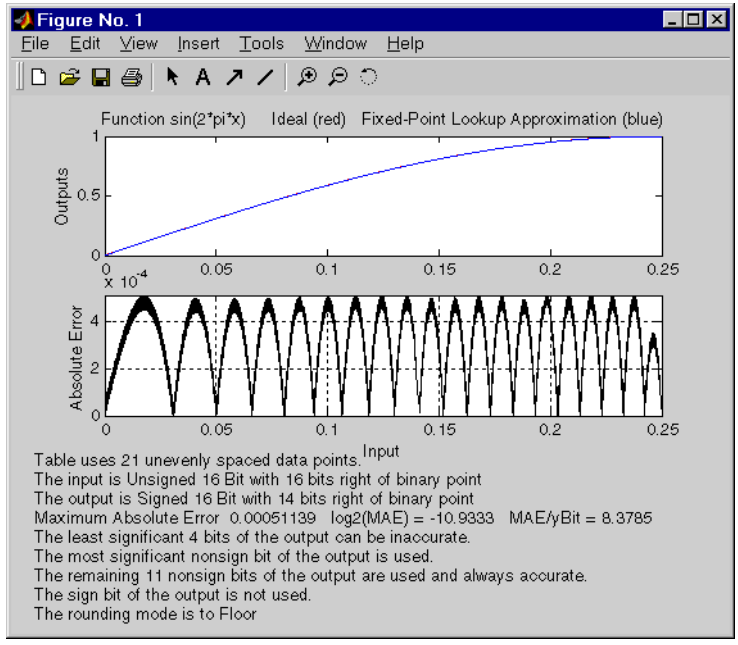
```
errworst =
  5.1139e-004
```

Plotting the Results

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(funcstr,xdata,xdt,xscale,ydata,ydt,...
yscale,rndmeth);
```

The resulting plots are shown below.



Restricting the Spacing

In the previous two examples, the function `fixpt_look1_func_approx` creates lookup tables with unrestricted spacing between the breakpoints. You can

restrict the spacing to improve the computational efficiency of the lookup table, using the spacing parameter.

The options for spacing are:

- `'unrestricted'` – The default.
- `'even'` – The distance between any two adjacent breakpoints is the same.
- `'pow2'` – The distance between any two adjacent breakpoints is the same and is a power of two.

Both power of two and even spacing increase the computational speed of the lookup table and use less command ROM. However, specifying either of the spacing restrictions along with `errmax` usually requires more data points in the lookup table than does unrestricted spacing, in order to achieve the same degree of accuracy. The section "Effect of Spacing on Speed, Error, and Memory Usage" on page 8-20 discusses the trade-offs between different spacing options.

Example 3: Using errmax with Even Spacing

The next example shows how to create a lookup table that has evenly spaced breakpoints and a specified worst case error. To try the example, you must first enter the enter the parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-6, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'even';
[xdata ydata errworst]=fixpt_look1_func_approx(funcstr,...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

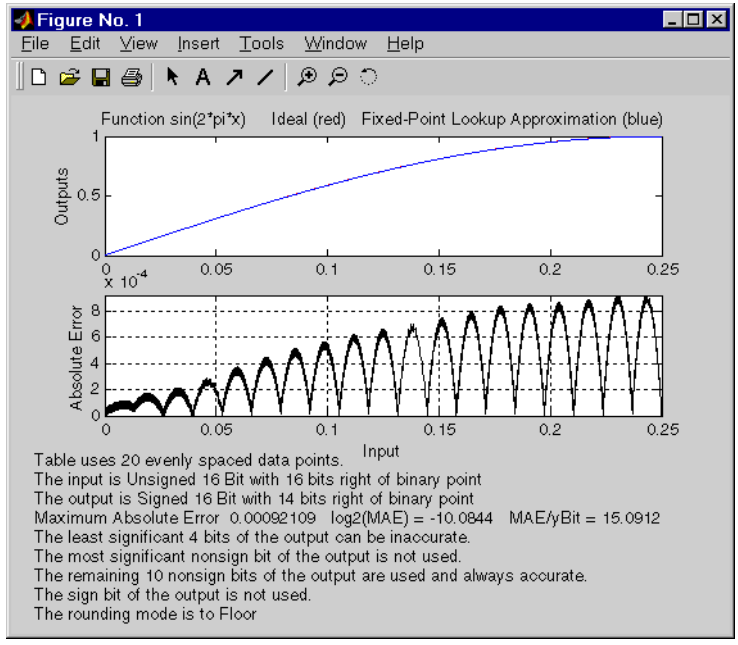
You can find the number of points in the lookup table by typing `length(xdata)`.

```
ans =
    20
```

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth);
```

This produces the following plots.



Example 4: Using nptsmax with Even Spacing

The next example shows how to create a lookup table that has evenly space breakpoints and minimizes the worst case error for a specified maximum number of points. To try the example, you must first enter the parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 8-6, if you have not already done so in this MATLAB session.

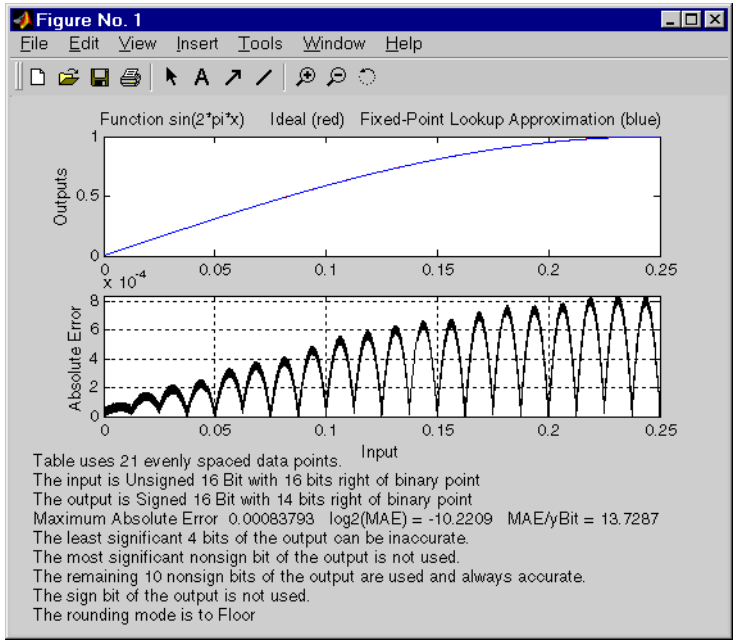
Next, type at the MATLAB prompt.

```
spacing='even';
[xdata ydata errworst]= fixpt_look1_func_approx(funcstr,...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax,spacing);
```

The result requires 21 evenly spaced points to achieve a maximum absolute error of `2^-10.2209`.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth);
```



Example 5: Using errmax with Power of Two Spacing

The next example shows how to construct a lookup table that has power of two spacing and a specified worst case error. To try the example, you must first enter the enter the parameter values given in the section “Setting Function Parameters for the Lookup Table” on page 8-6, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing ='pow2';
[xdata ydata
errworst]=fixpt_look1_func_approx(funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

To find out how many points are in the lookup table, type `length(xdata)`.

```
ans =
    33
```

This means that 33 points are required to achieve the worst case error specified by `errmax`. To verify that these points are evenly spaced, type

```
widths=diff(xdata)
```

This generates a vector whose entries are the differences between consecutive points in `xdata`. Every entry of `widths` is 2^{-7}.

To find the maximum error for the lookup table, type `errworst`.

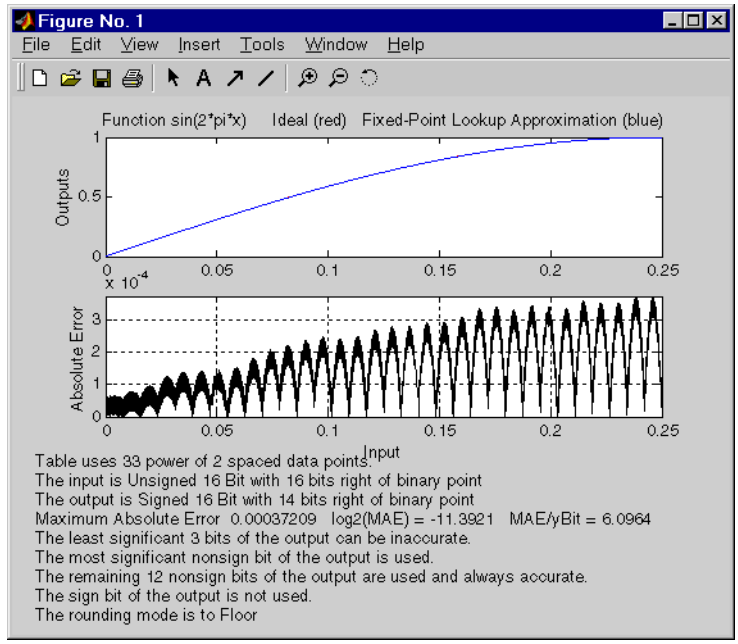
```
errworst =
  3.7209e-004
```

This is less than the value of `errmax`.

To plot the lookup table data along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth);
```

This displays the plots shown below.



Example 6: Using nptsmax with Power of Two Spacing

The next example shows how to create a lookup table that has power of two spacing and minimizes the worst case error for a specified maximum number of points. To try the example, you must first enter the enter the parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-6, if you have not already done so in this MATLAB session.

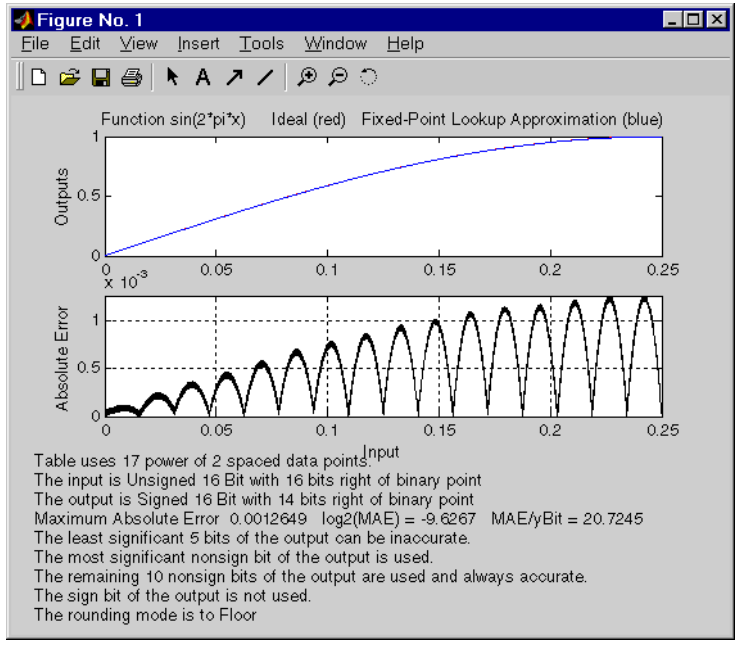
```
spacing ='pow2';
[xdata, errworst]= fixpt_look1_func_approx(funcstr,...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax,spacing);
```

The result requires 17 points to achieve a maximum absolute error of 2^-9.6267.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(funcstr,xdata,xdt,xscale,ydt,yscale,rndmeth);
```

This produces the plots shown below.



Specifying Both errmax and nptsmax

If you include both the `errmax` and the `nptsmax` parameters, the function `fixpt_look1_func_approx` tries to find a lookup table with at most `nptsmax` data points, whose worst case error is at most `errmax`. If it can find a lookup table meeting both conditions, it uses the following order of priority for spacing:

1. Power of two
2. Even
3. Unrestricted

If the function cannot find any lookup table satisfying both conditions, it ignores nptsmax and returns a lookup table with unrestricted spacing, whose worst case error is at most errmax. In this case, the function behaves the same as if the nptsmax parameter were omitted.

Using the parameters described the section "Setting Function Parameters for the Lookup Table" on page 8-6, the following examples illustrate the results of using different values for nptsmax when you enter

```
[xdata ydata errworst]=fixpt_look1_func_approx(funcstr,
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,numptsmax);
```

The results for three different settings for nptsmax are as follows:

- numptsmax=33 – The function creates the lookup table with 33 points having power of two spacing as in Example 3.
- numptsmax=21 – Since the errmax and numptsmax conditions cannot be met with power of two spacing, the function creates the lookup table with 20 points having even spacing, as in Example 5.
- numptsmax=16 – Since the errmax and numptsmax conditions cannot be met with either power of two or even spacing, the function creates the lookup table with 16 points having unrestricted spacing, as in Example 1.

Comparing the Examples

The following table summarizes the results for the examples.

Table 8-1: Summary of Results for the Examples

Example	Options		Worst Case Error	Number of Points in Table
1	errmax=2^-10	'unrestricted'	2^-10	16
2	nptsmax=21	'unrestricted'	2^-10.933	21
3	errmax=2^-10	'even'	2^-10.0844	20
4	nptsmax=21	'even'	2^-10.2209	21
5	errmax=2^-10	'pow2'	2^-11.3921	33
6	nptsmax=21	'pow2'	2^-9.627	17

Note that when you specify `errmax`, even spacing requires more data points than unrestricted, and power of two spacing requires more points than even spacing.

Summary of the Lookup Table Functions

The following summarizes how to use the lookup table approximation functions.

1. Define:
   - The ideal function to be approximated
   - The range, `xmin` to `xmax`, over which to find X and Y data
   - The fixed-point implementation: data type, scaling, and rounding method
   - The maximum acceptable error, the maximum number of points, and the spacing.
2. Run the `fixpt_look1_func_approx` function to generate X and Y data.
3. Use the `fixpt_look1_func_plot` function to plot the function and error between the ideal and approximated function using the selected X and Y data, and to calculate the error and the number of points used.
4. Vary input criteria, such as `errmax`, `nptsmax` and `spacing`, to produce sets of X and Y data that generate functions with varying worst-case error, number of points required, and spacing.
5. Compare results of the number of points required and maximum absolute error from various runs to choose the best set of X and Y data.

Effect of Spacing on Speed, Error, and Memory Usage

This section compares the implementations of lookup tables that use breakpoints whose spacing is uneven, even, and power of two. This comparison is only valid when the breakpoints are not tunable. If the breakpoints can be tuned in the generated code, then all three cases generate the same code. The comparison will focus on the amount of read only memory (ROM) used for data, the amount of ROM used for commands, and the speed with which the commands are executed.

As a specific example, this comparison uses the demo `fxpdemo_approx_sin`. There are three fixed-point lookup tables in this model. All three lookup tables approximate the function `sin(2*pi*u)` over the first quadrant. All three achieve a worst case error of less than `2^-8`. However, they have different restrictions on their breakpoint spacing.

You can use the model `fxpdemo_approx`, which this demo opens, to generate code with Real-Time Workshop. This section presents several segments of the generated code. These segments of code were edited and arranged for clarity and to emphasize key differences.

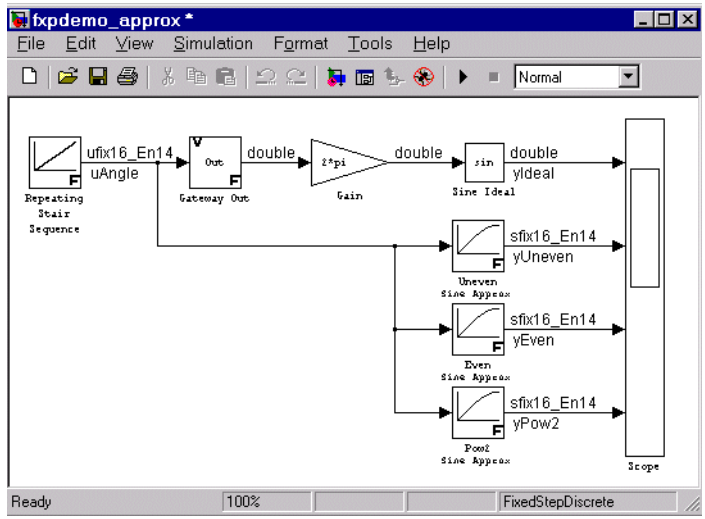
This section covers the following topics:

- "Data ROM Required" on page 8-21
- "Determining Out-of-Range Inputs" on page 8-22
- "Determining the Location of the Input" on page 8-22
- "Performing Interpolation." on page 8-24
- "Conclusion" on page 8-26

To open the demo, type at the MATLAB prompt

```
fxpdemo_approx_sin
```

This opens the model shown below.



Data ROM Required

This section looks at the data ROM required by each of the three spacing options.

Uneven Case

Uneven spacing requires both Y data points and breakpoints.

```
int16_T  yuneven[8];
uint16_T xuneven[8];
```

The total bytes used is 32.

Even Case

Even spacing requires only Y data points.

```
int16_T yeven[10];
```

The total bytes used is 20. The breakpoints are not explicitly required. The code will use the spacing between the breakpoints, and may use the smallest and largest breakpoint. At most three values related to the breakpoints are needed.

Power of Two Case

Power of two spacing requires only Y data points.

```
int16_T ypow2[17];
```

The total bytes used is 34. The breakpoints are not explicitly required. The code will use the spacing between the breakpoints, and may use the smallest and largest breakpoint. At most three values related to the breakpoints are needed.

Determining Out-of-Range Inputs

In all three cases you have to guard against the possibility that the input is less than the smallest breakpoint or greater than the biggest breakpoint. There may be differences in how occurrences of these possibilities are handled. However, the differences are generally minor and are normally not a key factor in deciding to use one spacing method over another. The subsequent sections assume that out-of-range inputs are impossible or have already been handled.

Determining the Location of the Input

This section describes how the three fixed point lookup tables determine where the current input is relative to the breakpoints.

Uneven Case

Unevenly spaced breakpoints require a general-purpose algorithm such as a binary search to determine where the input lies in relation to the breakpoints. The following code provides an example.

```
iLeft = 0;
iRght = 7; /* number of breakpoints minus 1 */

while ( ( iRght - iLeft ) > 1 )
{
  i = ( iLeft + iRght ) >> 1;

if ( uAngle < xuneven[i] )
  {
    iRght = i;
  }
  else
  {
```

```
      iLeft = i;
    }
  }
```

The while loop executes up to log2(N) times where N is number of breakpoints.

Even Case

Evenly spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints.

```
iLeft = uAngle / 455U;
```

The divisor `455U` represents the spacing between breakpoints. In general, the dividend would be `(uAngle - SmallestBreakPoint)`. In this example, the smallest breakpoint was zero, so the subtraction was optimized out.

Power of Two Case

Power of two spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints.

```
iLeft = uAngle >> 8;
```

The number of shifts is 8 because the breakpoints have spacing `2^8`. The smallest breakpoint was zero, so `uAngle` replaced the general case of `(uAngle - SmallestBreakPoint)`.

Comparison

To determine where the input is located with respect to the breakpoints, the unevenly spaced case clearly requires much more code than the other two cases. This code requires additional command ROM. This ROM penalty can be reduced if many lookup tables share the binary search algorithm as a function. Even if the code is shared, the number of clock cycles required to determine the location of the input is much higher for the unevenly spaced cases than the other two cases. If the code is shared, then function call overhead decreases the speed of execution a little more.

In the evenly spaced case and the power of two spaced case, you can determine the location of the input with a single line of code. The evenly spaced cased uses a general integer division. The power of two case uses a shift instead of general division because the divisor is an exact power of two. Without knowing the

specific processor to be used, you cannot be certain that a shift is better than division.

Many processors can implement division with a single assembly language instruction, so the code will be small. However, this instruction often takes many clock cycles to complete. Quite a few processors do not provide a division instruction. Division on these processors is implemented via repeated subtractions. This is slow and requires a fair amount of machine code, but this code can be shared.

Most processors provide a way to do logical and arithmetic shifts left and right. A distinguishing difference is whether the processor can do N shifts in one instruction (barrel shift) or requires N instructions that shift one bit at a time. The barrel shift will require less code. Whether or not the barrel shift also increases speed depends on the hardware that supports the operation.

The compiler can also complicate the comparison. In the previous example, the command `uAngle >> 8` essentially takes the upper 8 bits in a 16 bit word. The compiler may detect this and replace the bit shifts with an instruction that takes the bits directly. If the number of shifts is some other value, such as 7, this optimization would not occur.

Performing Interpolation.

In theory, you can calculate the interpolation with the following code.

```
y = ( yData[iRght] - yData[iLeft] ) * ( u - xData[iLeft] )
/ ( xData[iRght] - xData[iLeft] ) + yData[iLeft]
```

The term `(xData[iRght] - xData[iLeft])` is the spacing between neighboring breakpoints. If this value is constant, i.e., even spacing, some simplification is possible. If spacing is not just even but also a power of two, then very significant simplifications are possible for fixed-point implementations.

Uneven case

For the uneven case, one possible implementation of the ideal interpolation in fixed point is as follows.

```
xNum  = uAngle         - xuneven[iLeft];
xDen  = xuneven[iRght] - xuneven[iLeft];
yDiff = yuneven[iRght] - yuneven[iLeft];
```

```
MUL_S32_S16_U16( bigProd, yDiff, xNum );

  DIV_NZP_S16_S32_U16_FLOOR( yDiff, bigProd, xDen );

  yUneven = yuneven[iLeft] + yDiff;
```

The multiplication and division routines are not shown here. These can be somewhat involved and depend on the target processor. For example, these routines look quite different for a 16-bit processor than for a 32-bit processor.

Even Case

Evenly spaced breakpoints implement interpolation using just slightly different calculations than the uneven case. The key difference is that the calculations do not directly use the breakpoints. This means the breakpoints are not required in ROM, which can be a very significant savings.

```
xNum  = uAngle - ( iLeft * 455U );

  yDiff = yeven[iLeft+1] - yeven[iLeft];

  MUL_S32_S16_U16( bigProd, yDiff, xNum );

  DIV_NZP_S16_S32_U16_FLOOR( yDiff, bigProd, 455U );

  yEven = yeven[iLeft] + yDiff;
```

Power of Two Case

Power of two spaced breakpoints implement interpolation using very different calculations than the other two cases. Like the uneven case, breakpoints are not used in the generated code and therefore not required in ROM.

```
lambda = uAngle & 0x00FFU;

  yPow2 = ypow2[iLeft)+1] - ypow2[iLeft];

  MUL_S16_U16_S16_SR8(yPow2,lambda,yPow2);

  yPow2 += ypow2[iLeft];
```

This implementation has very significant advantages over the uneven and even implementations. The key difference is that a subtraction and a division are replaced by a bitwise-AND combined with a shift right at the end of the multiply. Another advantage is that the term `(u - xData[iLeft] ) / ( xData[iRght] - xData[iLeft])` is computed with no loss of precision, because the spacing is a power of two. In contrast, the uneven and even cases usually introduce rounding error in this calculation.

Conclusion

The number of Y data points follows the expected pattern. For the same worst case error, unrestricted spacing (uneven) requires the fewest data points, and power of two spaced breakpoints requires the most. However, the implementation for the evenly spaced and the power of two cases does not need the breakpoints in the generated code. This reduces their data ROM requirements by a half. As a result, the evenly spaced case actually uses less data ROM than the unevenly spaced case. Also, the power of two case requires only slightly more ROM than the uneven case. Changing the worst case error can change these rankings. Nonetheless, when you compare data ROM usage, you should always take into account the fact that the evenly spaced and power of two spaced cases do not require their breakpoints in ROM.

The effort of determining where the current input is relative to the breakpoints strongly favors the evenly spaced and power of two spaced cases. With uneven spacing, you use a binary search method that loops up to log2(N) times. With even and power of two spacing, you can determine the location with the execution of one line of C code. But you cannot decide the relative advantages of power of two versus evenly spaced without detailed knowledge of the hardware and the C compiler.

The effort of calculating the interpolation favors the power of two case, which uses a bitwise AND operation and a shift to replace a subtraction and a division. The amount of advantage provided by this depends on the specific hardware, but you would expect an advantage in code size, speed, and also in accuracy. The evenly space case calculates the interpolation with a minor improvement in efficiency over the unevenly spaced case.

9

Function Reference

Overview

This chapter contains reference pages for the Fixed-Point Blockset M-file functions. In some cases, you will not call these functions from the MATLAB command line. Instead, they are automatically called when you specify certain parameter values via block dialog boxes or via the Fixed-Point Blockset Interface tool. The functions are listed below.

Table 9-1: Fixed-Point Blockset Functions

Function	Description
`autofixexp`	Automatically change the scaling for each fixed- point block that does not have its scaling locked.
`fixptbestexp`	Determine the exponent that gives the best precision fixed-point representation of a value.
`fixptbestprec`	Determine the maximum precision available for the fixed-point representation of a value.
`fixpt_convert`	Convert Simulink models and subsystems to fixed-point equivalents.
`fixpt_convert_prep`	Prepare a Simulink model for more complete conversion to fixed point.
`fixpt_look1_func_approx`	Optimize for a fixed-point function, the x values that are generated for a lookup table.
`fixpt_look1_func_plot`	Plot a function with x values generated by the `fixpt_look1_func_approx` function.
`fixpt_restore_links`	Restore links for fixed-point blocks.
`float`	Create a MATLAB structure describing a floating-point data type.

Table 9-1: Fixed-Point Blockset Functions (Continued)

Function	Description
`fpupdate`	Update obsolete fixed-point blocks from previous Fixed-Point Blockset releases to current fixed-point blocks.
`fxptdlg`	Launch the Fixed-Point Blockset Interface tool.
`sfix`	Create a MATLAB structure describing a signed generalized fixed-point data type.
`sfrac`	Create a MATLAB structure describing a signed fractional data type.
`showfixptsimranges`	Display the logged maximum and minimum values from the last simulation.
`sint`	Create a MATLAB structure describing a signed integer data type.
`ufix`	Create a MATLAB structure describing an unsigned generalized fixed-point data type.
`ufrac`	Create a MATLAB structure describing an unsigned fractional data type.
`uint`	Create a MATLAB structure describing an unsigned integer data type.

autofixexp

Purpose

Automatically change the scaling for each fixed-point block that does not have its scaling locked

Syntax

```
autofixexp
```

Description

The `autofixexp` script automatically changes the scaling for each block that does not have its scaling locked. This script uses the maximum and minimum data obtained from the last simulation run to log data to the workspace. The scaling is changed such that the simulation range is covered and the precision is maximized. The script follows these steps:

1. The global variable `FixPtTempGlobal` is created to "steal" parameters (such as data type) from variables not known in the base workspace. For example, assume the Sum block has its output data type specified as `DerivedVar`. `DerivedVar` is derived in the mask initialization based on mask parameters and the block is under a mask.

   The value of the parameter `DerivedVar` is retrieved by temporarily replacing `DerivedVar` with `stealparameter(DerivedVar)` in the block dialog. A model update is then forced. When `stealparameter(DerivedVar)` is evaluated, it returns the value of `DerivedVar` without modification and stores the value in `FixPtTempGlobal`. The stolen value is immediately used by this procedure and is not needed again. Therefore, the procedure can move from one block to the next using the same global variable.

2. The `RangeFactor` variable allows you to specify a range differing from that defined by the maximum and minimum values logged in `FixPtSimRanges`. For example, a `RangeFactor` value of 1.55 specifies that a range *at least* 55 percent larger is desired. A value of 0.85 specifies that a range *up to* 15 percent smaller is acceptable.

   You should be aware that the scaling is not exact for the radix point-only case since the range is given (approximately) by a power of two. The lower limit is exact, but the upper limit is always one bit below a power of two.

   For example, if the maximum logged value is 5 and the minimum logged value is -0.5, then any `RangeFactor` from 4/5 to slightly under 8/5 would produce the same radix point since these limits are less than a factor of two from each other. The radix point selected will produce a range from -8 to +8 (minus a bit).

3. The global variable `FixPtSimRanges` is retrieved from the workspace. This is the variable that holds the maximum and minimum simulation values.
4. The workspace is searched for the variables `SlopeBits` and `BiasBits`, which specify the number of bits to use in representing slopes and biases. If these variables are not found, then they are automatically created with default values of 7 and 8, respectively.
5. All blocks that logged maximum and minimum simulation data are processed.
6. All blocks that do not have their scaling locked are automatically scaled. If the data type class is `FIX`, then radix point-only scaling is performed. If the data type class is `INT`, then slope/bias scaling is performed. To find out a data type's class, refer to its reference page in this chapter.

See Also

`fxptdlg`, `showfixptsimranges`

fixptbestexp

Purpose

Determine the exponent that gives the best precision fixed-point representation of a value

Syntax

```
out = fixptbestexp(RealWorldValue,TotalBits,IsSigned)
out = fixptbestexp(RealWorldValue,FixPtDataType)
```

Description

`out = fixptbestexp(RealWorldValue,TotalBits,IsSigned)` determines the exponent that gives the best precision for the fixed-point representation of the real world value specified by `RealWorldValue`. You specify the number of bits for the fixed-point number with `TotalBits`, and you specify whether the fixed-point number is signed with `IsSigned`. If `IsSigned` is `1`, the number is signed. If `IsSigned` is `0`, the number is not signed. The exponent is returned to `out`.

`out = fixptbestexp(RealWorldValue,FixPtDataType)` determines the exponent that gives the best precision based on the data type specified by `FixPtDataType`.

Example

The following command returns the exponent that gives the best precision for the real world value 4/3 using a signed, 16-bit number.

```
out = fixptbestexp(4/3,16,1)
out =
   -14
```

Alternatively, you can specify the fixed-point data type.

```
out = fixptbestexp(4/3,sfix(16))
out =
   -14
```

This value means that the maximum precision representation of 4/3 is obtained by placing 14 bits to the right of the binary point.

```
01.01010101010101
```

You would specify the precision of this representation in fixed-point blocks by setting the scaling to `2^-14` or `2^fixptbestexp(4/3,16,1)`.

See Also

`fixptbestprec`, `sfix`, `ufix`

Purpose

Determine the maximum precision available for the fixed-point representation of a value

Syntax

```
out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)
out = fixptbestprec(RealWorldValue,FixPtDataType)
```

Description

`out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)` determines the maximum precision for the fixed-point representation of the real world value specified by `RealWorldValue`. You specify the number of bits for the fixed-point number with `TotalBits`, and you specify whether the fixed-point number is signed with `IsSigned`. If `IsSigned` is `1`, the number is signed. If `IsSigned` is `0`, the number is not signed. The maximum precision is returned to `out`.

`out = fixptbestprec(RealWorldValue,FixPtDataType)` determines the maximum precision based on the data type specified by `FixPtDataType`.

Example

The following command returns the maximum precision available for the real world value 4/3 using a signed, 8-bit number.

```
out = fixptbestprec(4/3,8,1)
out =
   0.015625
```

Alternatively, you can specify the fixed-point data type.

```
out = fixptbestprec(4/3,sfix(8))
out =
   0.015625
```

This value means that the maximum precision available for 4/3 is obtained by placing six bits to the right of the binary point since 2^{-6} equals 0.015625.

```
01.010101
```

You can use the maximum precision as the scaling parameter in fixed-point blocks.

See Also

`fixptbestexp`, `sfix`, `ufix`

fixpt_convert

Purpose

Convert Simulink models and subsystems to fixed-point equivalents

Syntax

```
res = fixpt_convert
res = fixpt_convert('SystemName')
res = fixpt_convert('SystemName','Display')
res = fixpt_convert('SystemName','Display','AutoSave')
```

Description

`res = fixpt_convert` converts the Simulink model or subsystem specified by `bdroot`. `res` is a structure that contains lists of blocks handled during conversion. The fields of this structure are given below.

Output Field	Description
`replaced`	Blocks that are replaced with fixed-point equivalents or with other blocks from a user-specified replacement list.
`skipped`	Blocks that are skipped because they are fixed-point compatible. Some of these blocks can cause errors if used in certain ways. For example, the Mux block can create lines that give different data types at down stream input ports.
`encapsulated`	Structure containing lists of blocks grouped by type that are encapsulated between fixed-point gateway blocks. The encapsulated versions are not truly fixed point, but they will function within a fixed-point model.

`res = fixpt_convert('SystemName')` converts the Simulink model or subsystem specified by `SystemName`.

`res = fixpt_convert('SystemName','Display')` returns information associated with the conversion according to the method specified by *`Display`*. The *`Display`* methods are given below.

Display Method	Description
`on`	Display detailed block information.
`outline`	Display the conversion process outline.

Display Method	Description
`off`	Do not display block information.
`filename`	Write detailed block information to the specified file.
`on+filename`	Display detailed block information, and write detailed block information to the specified file.
`outline+filename`	Display the conversion process outline, and write detailed block information to the specified file.

`res = fixpt_convert('SystemName','Display','AutoSave')` determines the state of the converted model or subsystem. If *`AutoSave`* is `on`, then the converted model or subsystem is saved and closed. If *`AutoSave`* is `off`, then the converted model or subsystem is unsaved and left open.

Remarks

If your Simulink model references blocks from a custom Simulink library, then these blocks are encapsulated upon conversion. A block is encapsulated when it cannot be converted to an equivalent fixed-point block. Encapsulation involves associating a Gateway In or a Gateway Out block with the Simulink block. To reduce the number of blocks that are encapsulated, you should convert the entire library by passing the library name to `fixpt_convert`, and then convert the model.

To create a custom list of blocks to convert, you should use the `fixpt_convert_userpairs` script. To learn how to use this script, read the comments included in the M-file.

The data types for fixed-point outputs taking Boolean values are specified by the variable `LogicType`. The data types of all other fixed-point outputs and parameters are specified by the variable `BaseType`. You can change these variables to any data type. For example, in the MATLAB workspace you can type

```
BaseType = sfix(16)
LogicType = uint(8)
```

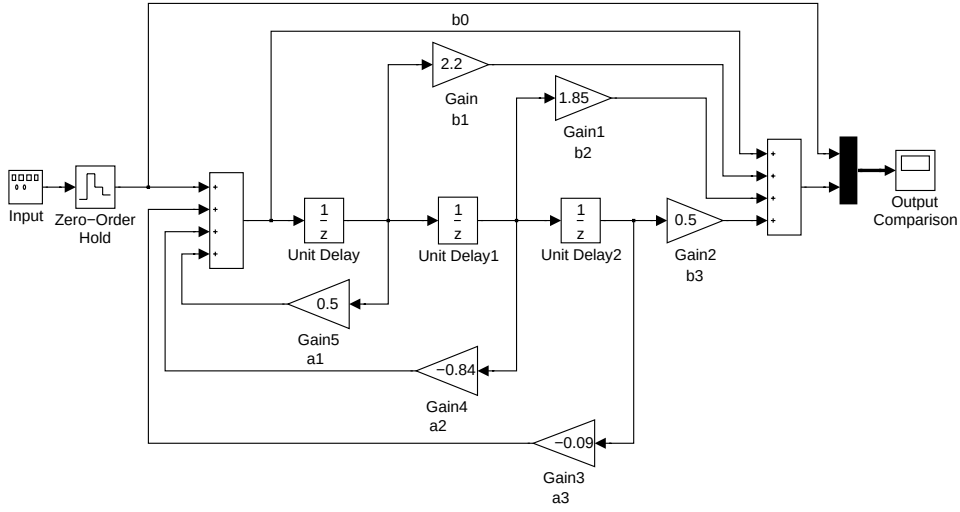
The converted model will not work if these variables are not defined.

Best precision mode is used when available. Otherwise, the precision is set to 2^0, which means that the binary point is to the right of all bits.To automatically

set the scaling, run a simulation with doubles override on and then invoke the automatic scaling script, `autofixexp`. You can run `autofixexp` directly, or in conjunction with the Fixed-Point Blockset Interface, `fxptdlg`.

Example

This example uses `fixpoint_convert` to convert a Simulink model of a direct form II realization to its fixed-point equivalent. This realization is discussed in "Direct Form II" on page 5-4. The simulink model shown below, `fxpdemo_preconvet`, is included as a demo with the blockset.



The following command converts this model to its fixed-point equivalent, suppresses the display of detailed block information, and does not save the model after conversion.

```
res = fixpt_convert('fxpdemo_preconvert','off','off')
```

The built-in blocks that are replaced by fixed-point equivalent blocks are given by the `replaced` field.

```
res.replaced
ans =
        UnitDelay: {3x1 cell}
    ZeroOrderHold: {[1x40 char]}
             Gain: {6x1 cell}
```

```
                Sum: {2x1 cell}
```

The built-in blocks that are skipped since they are compatible with the Fixed-Point Blockset are given by the `skipped` field.

```
res.skipped
ans =
Mux: {'fxpdemo_preconvert_fixpt/Mux'}
```

The built-in blocks that are encapsulated by fixed-point gateway blocks so that they are made compatible with the Fixed-Point Blockset are given by the `encapsulated` field.

```
res.encapsulated
ans =
               Scope: {[1x42 char]}
    SignalGenerator: {'fxpdemo_preconvert_fixpt/Input'}
```

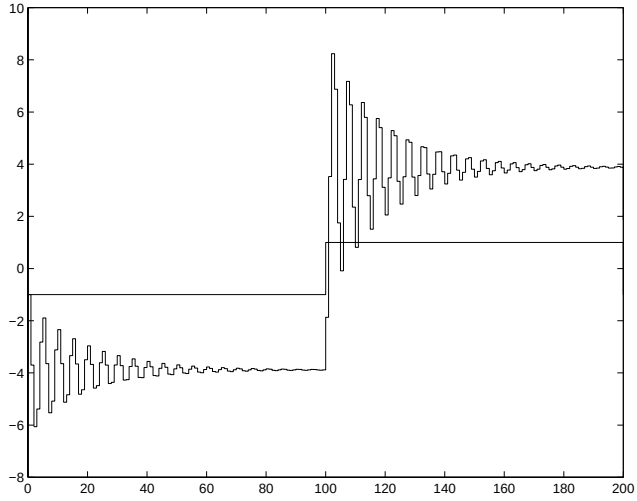
Note that the initial class of the base data type is `double`.

```
BaseType =
    Class: 'DOUBLE'
```

You can now run the simulation for the converted model.

```
sim fxpdemo_preconvert_fixpt
```

The output from the simulation is shown below. You should compare this output to the output produced by the fixed-point direct from II model, `fxpdemo_direct_form2`.



Next, define a fixed-point base data type.

```
BaseType = sfix(16)
```

Follow the automatic scaling procedure described in the `autofixexp` reference pages with 20% safety margin, and then run the simulation.

```
sim fxpdemo_preconvert_fixpt
```

The simulation now produces an error. This is because the vector signal leading into the scope is not homogeneous with regard to data type and scaling.

In general, solving the problem of nonhomogeneous signals requires that you analyze how the signal is being used. If the distinct scaling and data type properties are important, then you must fully or partially unvectorize the relevant part of the model. Alternatively, you can force the signals to be homogenous using the Gateway Out block. Since this example plots real world values in the Scope, inserting gateway blocks on the signals leading into the Scope is an adequate solution.

See Also

`autofixexp`, `fixpt_convert_prep`, `fxptdlg`

Purpose

Prepare a Simulink model for more complete conversion to fixed point

Syntax

```
fixpt_convert_prep('SystemName')
```

Description

`fixpt_convert_prep('SystemName')` prepares the Simulink model or subsystem specified by `SystemName` for more complete conversion (less encapsulation) to fixed point using the `fixpt_convert` function. It does so by replacing this select set of blocks:

- Old style Latch blocks

  Old style Latch blocks are replaced with a version contained in the `fixpt_convert_lib` library. The old style Latch block contains a Transport Delay block, which is a very inefficient implementation for both floating point and fixed point.
- Function blocks acting like selectors

  Function blocks acting like selectors are replaced with the Selector block. Function blocks acting like selectors require that you specify the width of the input. To get this information, the model must be put into compile mode, which is inefficient.
- A select set of additional function blocks

  You can replace function blocks that have replacements in the `fixpt_convert_lib` library. Alternatively, you can use `fixpt_convert_prep` as a prototype for creating a customized list of function blocks to be replaced. To do this, copy the function and the library to another directory, and then customize the library to include function blocks that you commonly encounter when converting models from floating point to fixed point.

Note This function is meant to be a starting point for customizing the Simulink to Fixed-Point Blockset conversion process.

See Also

`fixpt_convert`

fixpt_look1_func_approx

Purpose

Optimize for a fixed-point function, the x values, or breakpoints, that are generated for a lookup table

Syntax

```
[xdata,ydata,errworst]=fixpt_look1_func_approx('funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax)

[xdata,ydata,errworst]=fixpt_look1_func_approx('funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax)

[xdata,ydata,errworst]=fixpt_look1_func_approx('funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax)

[xdata,ydata,errworst]=fixpt_look1_func_approx('funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax,spacing)
```

Description

`fixpt_look1_func_approx('funcstr',xmin,xmax,xdt,xscale,ydt,yscale, rndmeth,errmax)` optimizes the breakpoints of a lookup table over a specified range. The lookup table satisfies the maximum acceptable error, maximum number of points, and spacing requirements given by the optional parameters. The breakpoints refer to the x values of the lookup table. The command

```
[xdata,ydata,errworst]=fixpt_look1_func_approx('funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax)
```

returns the X and Y coordinates of the lookup table as vectors `xdata` and `ydata`, respectively. It also returns the maximum absolute error of the lookup table as a variable `errworst`.

The fixed-point approximation is found by interpolating between the lookup table data points. The required input parameters are as follows.

Input	Value
`'funcstr'`	Function of x `funcstr` is the function for which breakpoints are approximated.
`xmin`	Minimum value of x
`xmax`	Maximum value of x
`xdt`	Data type of x

Input	Value
`xscale`	Scaling for the x values
`ydt`	Data type of y
`yscale`	Scaling for the y values
`rndmeth`	Rounding mode supported by the blockset: `'Toward Zero'`, `'Nearest'`, `'Floor'` (default value), `'Ceiling'`

- `xmin` and `xmax` specify the range over which the breakpoints are approximated.
- `xdt`, `xscale`, `ydt`, `yscale`, and `rndmeth` follow conventions used by the Fixed-Point Blockset.
- `rndmeth` has a default value listed in the input table.

In addition to the required parameters, there are three optional inputs, as follows.

.

Input	Value
`errmax`	Maximum acceptable error
`nptsmax`	Maximum number of points
`spacing`	Spacing: `'even'`, `'pow2'` (even power of 2), `'unrestricted'` (default value)

Of these, you must use at least one of the parameters `errmax` and `nptsmax`. If you omit one of these, use brackets, `[]`, in place of the omitted parameter. The function will then ignore that requirement for the lookup table.

The outputs of the function are as follows.

Output	Value
`xdata`	The breakpoints for the lookup table
`ydata`	The ideal function applied to the breakpoints
`errworst`	The worst case error, which is the maximum absolute error between the ideal function and the approximation given by the lookup table

Criteria For Optimizing the Breakpoints: errmax, nptsmax, and spacing

The approximation produced from the lookup table must satisfy the requirements for the maximum acceptable error, `errmax`, the maximum number of points, `nptsmax`, and the spacing, `spacing`. The requirements are:

- The maximum absolute error is less than `errmax`.
- The number of points required is less than `nptsmax`.
- The spacing is specified as unrestricted, even or even power of 2.

Modes for errmax, nptsmax, and spacing

- If both `errmax` and `nptsmax` are specified.

  The returned breakpoints will meet both criteria if possible. The `errmax` parameter is given priority, and `nptsmax` is ignored, if both criteria cannot be met with the specified spacing.
- If only `errmax` is specified.

  The breakpoints that meet the error criteria, and have the least number of points are returned.
- If only `nptsmax` is specified.

  The breakpoints that require `nptsmax` or fewer, and give the smallest worst case error are returned
- If more than one spacing method meets the criteria, power of 2 spacing is chosen over even spacing, which in turn is chosen over uneven spacing. This

case occurs when the `errmax` and `nptsmax` are both specified, but does not typically occur when only one is specified.

If `unrestricted` is entered, the function chooses the spacing that provides the best optimization.

If `even` is entered, the function chooses an evenly spaced set of points, including the `pow2` spacing.

If `pow2` spacing is entered, the function chooses an even power of 2 spaced set of points.

Note The global optimum may not be found. The worst case error can depend on fixed-point calculations, which are highly nonlinear. Furthermore, the optimization approach is heuristic.

Remarks

Using the Approximation Function

1. Choose a function and use the `eval('funcstr');` command to view the function before creating the lookup table.
2. Define the remaining inputs.
3. Run the `fixpt_look1_func_approx` function.
4. Use the `fixpt_look1_func_plot` to plot the function from the selected breakpoints, and to calculate the error and the number of points used.
5. Vary the inputs to produce sets of breakpoints that generate functions with varying number of points required and worst case error.
6. Compare the number of points required and worst case error from various runs to choose the best set of breakpoints.

Calculating the Output Function

To calculate the function, use the returned breakpoints with:

- The `eval` function
- A function lookup table. The x values are the breakpoints from the `fixpt_look1_func_approx` function, and the y values can be supplied using the `eval` function.

Selecting Spacing to Optimize Speed, Error, ROM, and RAM

The spacing you choose depends on the parameters you want to optimize: execution speed, function approximation error, ROM usage, and RAM usage.

- The execution speed depends on the bisection search, and the interpolation method.
- The error depends on how accurately the method approximates the nonuniform curvature of the function.
- The ROM usage depends on the amount of data and command ROM used.
- The RAM usage depends on how much global and stack RAM is used

When the lookup table has even power of two spacing, division is replaced by a bit shift. As a result, the execution speed is faster than for evenly spaced data.

See the chapter Chapter 8, “Producing Lookup Table Data” for a tutorial on using `fixpt_look1_func_approx`.

The following table summarizes the effect of spacing on the execution speed, error, and memory used.

Table 9-2: Comparison of the Spacing Options

Parameter	Even Power of Two Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
Execution Speed	The execution speed is the fastest. The position search and interpolation are the same as for evenly spaced data. However, to increase the speed more, the position search is replaced by a bit shift, and the interpolation is replaced with a bit mask.	The execution speed is faster then that for unevenly spaced data because the position search is faster and the interpolation requires a simple division.	The execution speed is the slowest of the different spacings because the position search is slower, and the interpolation requires more operations.
Error	The error can be larger than that for unevenly spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be larger than that for unevenly spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be smaller because approximating a function with nonuniform curvature requires fewer points to achieve the same accuracy.
ROM Usage	Uses less command ROM, but more data ROM.	Uses less command ROM, but more data ROM.	Uses more command ROM, and less data ROM.
RAM Usage	Not significant.	Not significant.	Not significant.

Example

The next example produces a lookup table for a sine function. The inputs for the example are as follows.

```
funcstr = 'sin(2*pi*x)';
xmin = 0;
xmax = 0.25;
xdt = ufix(16);
xscale = 2^-16;
ydt = sfix(16);
yscale = 2^-14;
rndmeth = 'Floor';
errmax = 2^-10;
spacing = 'pow2';
```

To create the lookup table, type

```
[xdata, ydata, errWorst]=fixpt_look1_func_approx(funcstr,
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

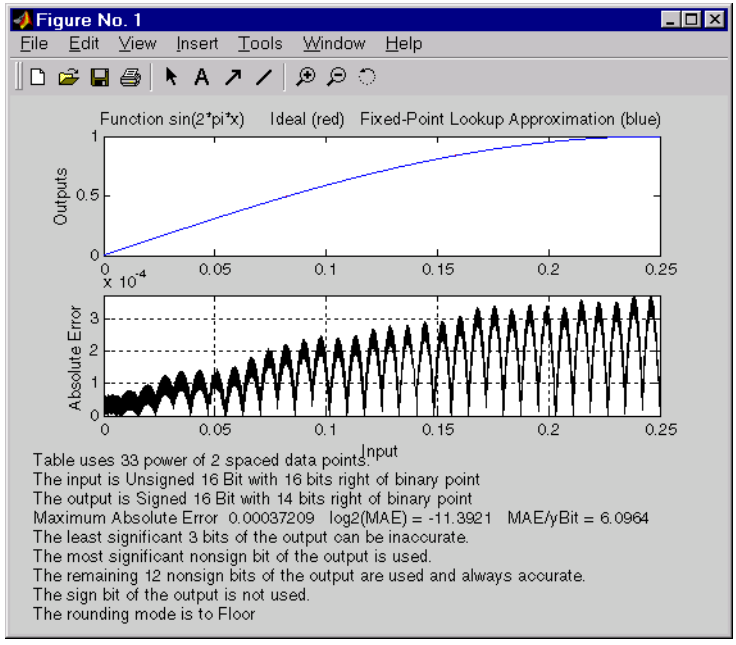
The brackets [ ] are a place holder for the `nptsmax` parameter, which is not used in this example.

You can then plot the ideal function, the approximation, and the errors by typing

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt,...
xscale,ydt,yscale,rndmeth);
```

The `fixpt_look1_func_plot` function produces a plot of the fixed-point sine function, using these breakpoints, and a plot of the error between the ideal function and the fixed-point function. The maximum absolute error and the number of points required are listed with the plot. The error drops to zero at a breakpoint, and increases between breakpoints due to the difference in curvature of the ideal function and the line drawn between breakpoints.

The resulting plots are shown below.



The lookup table requires 33 points to achieve a maximum absolute error of `2^-11.3922`

See Also

`fixpt_look1_func_plot`

fixpt_look1_func_plot

Purpose

Plot a function with x values generated by the `fixpt_look1_func_approx` function

Syntax

```
errworst=fixpt_look1_func_plot(xdata,ydata,'funcstr',...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

Description

`fixpt_look1_func_plot(xdata,ydata,'funcstr',xmin,xmax,xdt,xscale, ydt,yscale,rndmeth)` plots a lookup table approximation function and its error from the ideal function. You can use the `fixpt_look1_func_approx` function to generate `xdata` and `ydata`, the X and Y data points for the lookup table. The function returns the maximum absolute error as a variable `errworst`. The inputs are as follows.

Input	Value
`xdata`	x values for the lookup table
`ydata`	y values for the lookup table
`'funcstr'`	Function of x
`xmin`	Minimum input of interest
`xmax`	Maximum input of interest
`xdt`	Data type of x
`xscale`	Scaling for the x values
`ydt`	Data type of y
`yscale`	Scaling for the y values
`rndmeth`	Rounding mode supported by the blockset: `'Toward Zero'`, `'Nearest'`, `'Floor'`, `'Ceiling'`

The `fixpt_look1_func_approx` function applies the ideal function to the points in `xdata` to produce `ydata`. While this is is the easiest way to generate `ydata`, you are not required to use these values for `ydata` as input for the

`fixpt_look1_func_approx` function. Choosing different values for `ydata` can, in some cases, produce a lookup table with a smaller maximum absolute error.

See Chapter 8, “Producing Lookup Table Data” for a tutorial on using the function `fixpt_look1_func_plot`. For an example of the function, see the reference page for the function fixpt_look1_func_approx on page 9-14.

See Also

`fixpt_look1_func_approx`

fixpt_restore_links

Purpose

Restore links for fixed-point blocks

Syntax

```
res = fixpt_restore_links
res = fixpt_restore_links('SystemName')
res = fixpt_restore_links('SystemName','AutoSave')
```

Description

`res = fixpt_restore_links` restores broken links for the fixed-point blocks contained in the model or subsystem specified by `bdroot`. By default, the models and libraries containing restored block links are left open and unsaved. `res` contains the names of the blocks that had broken links restored.

`res = fixpt_restore_links('SystemName')` restores links for the fixed-point blocks contained in the model or subsystem specified by `SystemName`.

`res = fixpt_restore_links('SystemName','AutoSave')` determines the state of the models or subsystems containing restored block links. If *`AutoSave`* is `on`, the models or subsystems are saved and closed. If *`AutoSave`* is `off`, the models or subsystems are unsaved and left open.

Remarks

Breaking library links to fixed-point blocks will almost certainly produce an error when you attempt to run the model. If broken links exist, you will likely uncover them when upgrading to the latest release of the Fixed Point Blockset.

Purpose

Create a MATLAB structure describing a floating-point data type

Syntax

```
a = float('single')
a = float('double')
a = float(TotalBits, ExpBits)
```

Description

`float('single')` returns a MATLAB structure that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`float('double')` returns a MATLAB structure that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

`float(TotalBits, ExpBits)` returns a MATLAB structure that describes a nonstandard floating-point data type that mimics the IEEE style. That is, the numbers are normalized with a hidden leading one for all exponents except the smallest possible exponent. However, the largest possible exponent might not be treated as a flag for Inf's and NaN's.

`float` is automatically called when a floating point number is specified in a block dialog box.

Note Unlike fixed-point numbers, floating point numbers are not subject to any specified scaling.

Example

Define a nonstandard, IEEE-style, floating-point data type with 31 total bits (excluding the hidden leading one) and 9 exponent bits.

```
a = float(31,9)
a =
        Class: 'FLOAT'
     MantBits: 21
      ExpBits: 9
```

See Also

`sfix`, `sfrac`, `sint`, `ufix`, `ufrac`, `uint`

fpupdate

Purpose

Update obsolete fixed-point blocks from previous Fixed-Point Blockset releases to current fixed-point blocks

Syntax

```
fpupdate('model')
fpupdate('model',blkprompt)
fpupdate('model',blkprompt,varprompt)
fpupdate('model',blkprompt,varprompt,muxprompt)
fpupdate('model',blkprompt,varprompt,muxprompt,message)
```

Description

`fpupdate('model')` replaces all obsolete fixed-point blocks contained in `model` with current fixed-point blocks. The model must be opened prior to calling `fpupdate`.

`fpupdate('model',blkprompt)` prompts you for replacement of obsolete blocks. If `blkprompt` is 0 (the default), you will not be prompted. If `blkprompt` is 1, you will have these three options:

- `y` (default) replaces the block.
- `n` does not replace the block.
- `a` replaces all blocks without further prompting.

`fpupdate('model',blkprompt,varprompt)` gives you the option of updating variables which appear in each block's dialog box with their actual numerical values. Note that such an update is possible only if the variables can be evaluated in the MATLAB workspace. If `varprompt` is 1 (the default), you are prompted for each variable found in the block diagram. If `varprompt` is 0, all variables are automatically updated without prompting.

`fpupdate('model',blkprompt,varprompt, muxprompt)` allows you to update the input size parameters of the Mux and Demux blocks found in `model`. The input sizes of these blocks may need to be updated to account for the mismatch between the old and new fixed-point data representations. In the old representation, each number had a width of 2. In the new representation, each number has a width of 1. To update Mux and Demux blocks that have only fixed-point inputs, the vector that specifies the input size should be divided by 2. If `muxprompt` is 1 (the default), each Mux and Demux block found in `model` is updated. If `muxprompt` is 0, the Mux and Demux blocks are automatically updated without prompting.

`fpupdate('model',blkprompt,varprompt,muxprompt,message)` allows you to show or suppress any warning or update messages generated during the update process. If `message` is 1 (the default), all messages are displayed. If `message` is 0, all messages are suppressed.

`fpupdate` calls `addterms` to terminate any unconnected input or output ports by attaching Ground or Terminator blocks, respectively.

Example

To see how `fpupdate` works, convert the obsolete model `fixpoint/obsolete/fpex1.mdl`.

```
fpex1
fpupdate('fpex1')
```

fxptdlg

Purpose

Launch the Fixed-Point Blockset Interface tool

Syntax

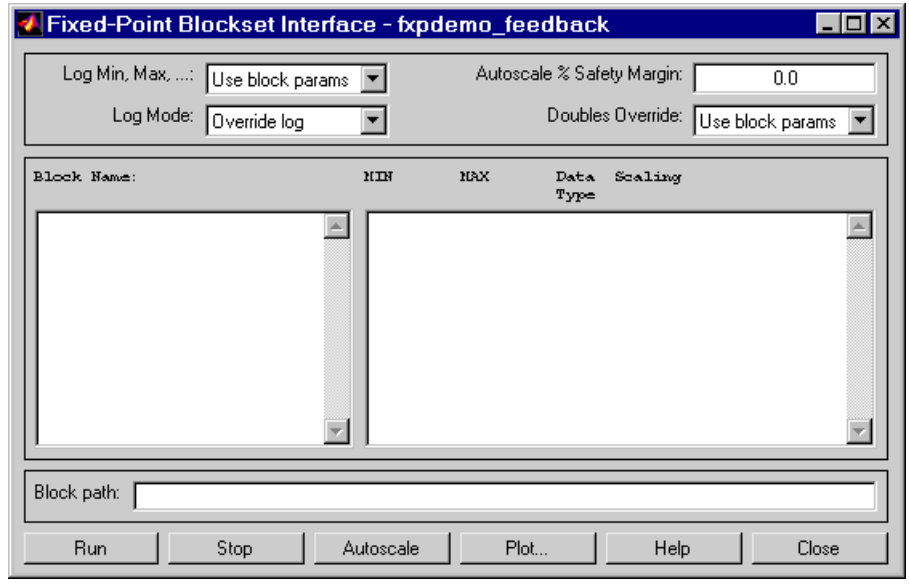
```
fxptdlg('model')
```

Description

`fxptdlg('model')` launches the Fixed-Point Blockset Interface tool for the fixed-point MDL-file `model`. The interface provides convenient access to the global overrides and min/max logging settings, the logged min/max data, the automatic scaling script, and the plot interface tool. You can launch the Interface tool for as many different MDL-files as you want, and the tool controls only the specified model. You can also invoke the Interface tool from the **Tools** menu in the model window, or with the Fixed-Point GUI block, which is included with all blockset demos.

For each block in the model that logs data, the Interface tool displays the block names, the minimum simulation value, the maximum simulation value, the data type, and the scaling. Additionally, if a signal saturates or overflows, then a message is displayed for the associated block indicating how many times saturation or overflow occurred. You can display a block's dialog box by double-clicking on the appropriate entry.

Parameters and Dialog Box



The Log Min/Max

This menu controls which blocks log data. `All` logs min/max data for all blocks, `None` doesn't log any min/max data, and `Use block params` logs min/max data for all blocks that have the **Log minimums and maximums** check box checked.

Log Mode

This menu controls how the log file is updated when multiple simulations are run. `Override log` updates all logged values for each simulation run. `Merge log` keeps the highest and lowest logged values across multiple simulations.

Autoscale % Safety Margin

This parameter multiplies the simulation values by the specified factor, and allows you to specify a range differing from that defined by the maximum and minimum values logged to the workspace. For example, a value of 55 specifies that a range *at least* 55 percent larger is desired. A value of -15 specifies that a range *up to* 15 percent smaller is acceptable.

The **Autoscale % Safety Margin** parameter is used as part of the automatic scaling procedure. Before automatic scaling is performed, you must run the simulation to collect min/max data.

Data Type Override

This menu controls overriding the output data type for all blocks in a system. `Doubles override on` overrides for all blocks the integer part of the fixed-point number with doubles. `Doubles override off` turns off the **Override data type(s) with doubles** for all blocks, so that the data type cannot be overwritten with doubles if the check box is checked. `Use block params` overrides the output data type with doubles for blocks that have the **Override data type(s) with doubles** check box checked. `True doubles` overrides for all blocks the output data type with doubles. `True singles` overrides for all blocks the data type with singles for all blocks.

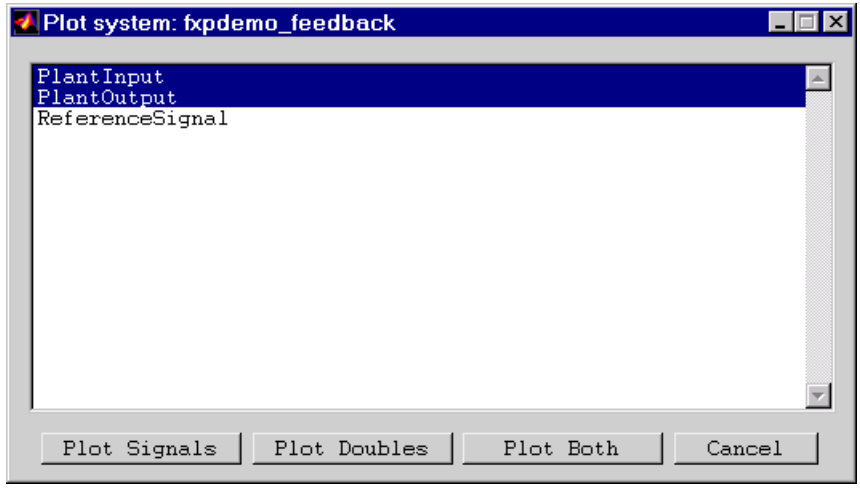
Block path

Displays the path for each selected block. The block path is described in terms of the blockset model name and, if required, the subsystem names.

The Fixed-Point Blockset Interface tool contains six buttons: **Run**, **Stop**, **Autoscale**, **Plot**, **Help**, and **Close**. The **Run** button runs the model and updates

the display with the latest simulation information. The **Stop** button stops the simulation from running. The **Autoscale** button invokes the automatic scaling script `autofixexp`. The **Plot** button invokes the Plot System interface, which displays any To Workspace, Outport, or Scope blocks found in the model. The **Help** button displays the HTML-based help. The **Close** button closes the Interface tool.

The Plot System interface is shown below. It is displaying Scope block output from the `fxpdemo_feedback` demo.



To plot the simulation results, select one or more variable names, and then select the appropriate plot button. You plot the raw signal data with the **Plot Signals** button. Raw signal data is generated when the global override switch is off. You plot doubles with the **Plot Doubles** button. Doubles are generated when the global override switch is on. You can plot both raw signal data and doubles with the **Plot Both** button. Note that the doubles override does not overwrite the raw data.

Example

To learn how to use the Fixed-Point Blockset Interface tool, refer to “Simulation Results” on page 6-9.

See Also

`autofixexp`, `showfixptsimranges`

Purpose

Create a MATLAB structure describing a signed generalized fixed-point data type

Syntax

```
a = sfix(TotalBits)
```

Description

`sfix(TotalBits)` returns a MATLAB structure that describes the data type of a signed generalized fixed-point number with a word size given by `TotalBits`.

`sfix` is automatically called when a signed generalized fixed-point data type is specified in a block dialog box.

Note A default radix point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

Example

Define a 16-bit signed generalized fixed-point data type.

```
a = sfix(16)
a =
       Class: 'FIX'
    IsSigned: 1
    MantBits: 16
```

See Also

`float`, `sfrac`, `sint`, `ufix`, `ufrac`, `uint`

sfrac

Purpose

Create a MATLAB structure describing a signed fractional data type

Syntax

```
a = sfrac(TotalBits)
a = sfrac(TotalBits, GuardBits)
```

Description

`sfrac(TotalBits)` returns a MATLAB structure that describes the data type of a signed fractional number with a word size given by `TotalBits`.

`sfrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of a signed fractional number. The total word size is given by `TotalBits` with `GuardBits` bits located to the left of the sign bit.

`sfrac` is automatically called when a signed fractional data type is specified in a block dialog box.

The default radix point for this data type is assumed to lie immediately to the right of the sign bit. If guard bits are specified, they lie to the left of the radix point in addition to the sign bit.

Example

Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this number is $-2^4 = -16$ to $(1 - 2^{(1 - 8)}) \cdot 2^4 = 15.875$.

```
a = sfrac(8,4)
a =
        Class: 'FRAC'
     IsSigned: 1
     MantBits: 8
    GuardBits: 4
```

See Also

`float`, `sfix`, `sint`, `ufix`, `ufrac`, `uint`

Purpose

Display the logged maximum and minimum values from the last fixed-point simulation.

Description

The `showfixptsimranges` script displays the logged maximum and minimum values from the last fixed-point simulation. Data is logged only from blocks where the **Log minimums and maximums** check box is checked.

The logged data is stored in the `FixPtSimRanges` cell array, which can be accessed by the `autofixexp` automatic scaling script.

See Also

`autofixexp`, `fxptdlg`

sint

Purpose

Create a MATLAB structure describing a signed integer data type

Syntax

```
a = sint(TotalBits)
```

Description

`sint(TotalBits)` returns a MATLAB structure that describes the data type of a signed integer with a word size given by `TotalBits`.

`sint` is automatically called when a signed integer is specified in a block dialog box.

The default radix point for this data type is assumed to lie to the right of all bits.

Example

Define a 16-bit signed integer data type.

```
a = sint(16)
a =
       Class: 'INT'
    IsSigned: 1
    MantBits: 16
```

See Also

`float`, `sfix`, `sfrac`, `ufix`, `ufrac`, `uint`

Purpose

Create a MATLAB structure describing an unsigned generalized fixed-point data type

Syntax

```
a = ufix(TotalBits)
```

Description

`ufix(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned generalized fixed-point data type with a word size given by `TotalBits`.

`ufix` is automatically called when an unsigned generalized fixed-point data type is specified in a block dialog box.

Note The default radix point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

Example

Define a 16-bit unsigned generalized fixed-point data type.

```
a = ufix(16)
a =
       Class: 'FIX'
    IsSigned: 0
    MantBits: 16
```

See Also

`float`, `sfix`, `sfrac`, `sint`, `ufrac`, `uint`

ufrac

Purpose

Create a MATLAB structure describing an unsigned fractional data type

Syntax

```
a = ufrac(TotalBits)
a = ufrac(TotalBits, GuardBits)
```

Description

`ufrac(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number with a word size given by `TotalBits`.

`ufrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number. The total word size is given by `TotalBits` with `GuardBits` bits located to the left of the radix point.

`ufrac` is automatically called when an unsigned fractional data type is specified in a block dialog box.

The default radix point for this data type is assumed to lie immediately to the left of all bits. If guard bits are specified, then they lie to the left the default radix point.

Example

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this number is from 0 to $(1 - 2^{-8}) \cdot 2^4 = 15.9375$.

```
a = ufrac(8,4)
a =
        Class: 'FRAC'
     IsSigned: 0
     MantBits: 8
    GuardBits: 4
```

See Also

`float`, `sfix`, `sfrac`, `sint`, `ufix`, `uint`

Purpose

Create a MATLAB structure describing an unsigned integer data type

Syntax

```
a = uint(TotalBits)
```

Description

`uint(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned integer with a word size given by `TotalBits`.

`uint` is automatically called when an unsigned integer is specified in a block dialog box.

The default radix point for this data type is assumed to lie to the right of all bits.

Example

Define a 16-bit unsigned integer.

```
a = uint(16)
a =
        Class: 'INT'
    IsSigned: 0
    MantBits: 16
```

See Also

`float`, `sfix`, `sfrac`, `sint`, `ufix`, `ufrac`

10

Block Reference

The Block Reference Pages 10-2

The Block Dialog Box 10-3

Common Block Features 10-4
Block Parameters 10-4
Block Icon Labels 10-10
Port Data Type Display 10-10

The Fixed-Point Blockset Libraries 10-12

The Block Reference Pages

Fixed-Point Blockset block reference pages appear in alphabetical order and contain some or all of this information:

- The block name and icon
- The purpose of the block
- A description of the block
- Additional remarks about block usage
- The block parameters and dialog box including a brief description of each parameter
- The rules for some or all of these topics, as they apply to the block:
  - Converting block parameters from double precision numbers to Fixed-Point Blockset data types
  - Converting the input data type(s) to the output data type
  - Performing block operations between inputs and parameters
- An example using the block
- The block characteristics, including some or all of these, as they apply to the block:
  - Input Port(s) – the data type(s) accepted by the block and whether the inputs can be a scalar or vector
  - Output Port – the data type(s) produced by the block and whether the outputs can be a scalar or vector
  - Direct Feedthrough – whether the block or any of its ports has direct feedthrough
  - Sample Time – how the block's sample time is determined, whether by the block itself or inherited from the block that drives it or is driven by it
  - Scalar Expansion – whether or not scalars are expanded to vectors
  - States – the number of discrete states
  - Vectorized – whether or not the block accepts and/or generates vector signals

The Block Dialog Box

You configure Fixed-Point Blockset blocks with a parameter dialog box. The parameter dialog box provides you with:

- The name and block type at the top of the dialog box
- A brief description of the block's behavior below the title
- Zero or more editable parameter fields, check boxes, or parameter lists below the description. You specify the parameter values using valid MATLAB expressions.
- A row of four buttons labeled **OK**, **Cancel**, **Help,** and **Apply** at the bottom of the dialog box. The **OK** button sets the current parameter values and closes the dialog box. The **Cancel** button reverts all the parameter values back to their values at the time the dialog box was opened, losing any changes you made. The **Help** button displays the HTML-based reference information. The **Apply** button sets the current parameter values and but does not close the dialog box.

Simulink stores the strings entered in these fields and passes them to MATLAB for evaluation when a simulation is started. If MATLAB variables are used, the simulation uses the values that exist in the workspace at the start of the simulation. These variables are not necessarily the same as when the variables are entered into the dialog box fields. If a simulation is running when a parameter is changed, MATLAB evaluates the parameter as soon as you press the **OK** or **Apply** button.

Common Block Features

For convenience, all the common block features are described in this section. These common features include:

- Block parameters
- Block icon labels
- Port data type display

Block Parameters

Many Fixed-Point Blockset blocks use the same parameters, which you configure through the block dialog box. The common block parameters are associated with these blockset features:

- Parameter and output data type selection
- Parameter and output scaling selection
- Rounding
- Overflow handling
- Overriding the output data type with doubles
- Logging simulation results

Block-specific parameters are described in the block reference pages.

Selecting the Data Type

For many fixed-point blocks, you need to associate data type information with numerical parameters and the output. You can associate data type information in these ways:

- **Parameters**

  The numerical parameter values of some fixed-point blocks inherit the data type of an input signal or the output signal. Other blocks require that you specify the parameter data type explicitly with the **Parameter data type** parameter.

- **Output**

  The output of some fixed-point blocks inherits the data type of the input signal. Other blocks require that you specify the output data type with the **Output data type** parameter. Still other blocks provide you with the option

of inheriting the output data type (and scaling) information from a driving block, or specifying the data type.

For the latter case, you control how the output data type (and scaling) is specified with the **Output data type and scaling** parameter list. This list supports three choices: `Specify via dialog`, `Inherit via internal rule`, and `Inherit via back propagation`. The parameter choices involving data type inheritance are designed to minimize specification burden. Note that some fixed-point blocks support only two of the three choices.

If you select `Specify via dialog`, you must explicitly specify the output data type with the **Output data type** parameter.

If you select `Inherit via internal rule`, the output data type is inherited from the input(s). The goal of the inheritance rule is to select the "natural" data type and scaling for the output. The specific rule that is used depends on the block operation. For example, if you are multiplying two signed 16-bit signals, the Product block produces the natural output of a signed 32-bit data type. An "unnatural" output is produced if the inputs have different signs and different sizes. In this case, some trial and error may be required to achieve satisfactory results. If you are adding signals, two natural choices for the output data type and scaling are possible: to preserve the precision or to prevent overflow. However, blocks only support one rule. For example, the Sum block preserves precision. If your goal is to prevent overflow, then you should manually configure the data type and scaling.

If you select `Inherit via back propagation`, the output data type is inherited by back propagation. In many cases, you will find that the Data Type Propagation block provides you with the most flexibility when back propagating the data type.

The supported data types and default scaling are shown below.

Table 10-1: Output Data Types and Default Scaling

Data Type	Description	Default Scaling
`float`	Floating-point number	None
`ufix`	Unsigned generalized fixed-point number	None

Table 10-1: Output Data Types and Default Scaling (Continued)

Data Type	Description	Default Scaling
`sfix`	Signed generalized fixed-point number	None
`uint`	Unsigned integer	Right of the least significant bit
`sint`	Signed integer	Right of the least significant bit
`ufrac`	Unsigned fractional number	Left of the most significant bit
`sfrac`	Signed fractional number	Right of the sign bit

The word size (in bits) of fixed-point data types is given as an argument to the data type. For example, `sfix(16)` specifies a 16-bit signed generalized fixed-point number. Word sizes from 1 to 128 bits are supported.

Floating-point data types are IEEE-style and are specified as `float('single')` for single-precision numbers and `float('double')` for double-precision numbers. Nonstandard IEEE-style numbers are specified as `float(TotalBits,ExpBits)` where `TotalBits` is the total number of physical bits and `ExpBits` is the number of exponent bits.

Note A default radix point is not included with the generalized fixed-point data type. Instead, the scaling must be explicitly specified as described below.

For more information about supported data types and their default scaling, refer to Chapter 3, “Data Types and Scaling.”

Selecting the Scaling

For generalized fixed-point data types, you need to associate scaling information with numerical parameters and the output. You can associate scaling information in these ways:

- **Parameters**

  The numerical parameter values of some fixed-point blocks inherit the scaling of an input signal or the output signal. Other blocks require that you specify the parameter scaling explicitly with the **Parameter scaling** parameter.

- **Output**

  The output of some fixed-point blocks inherits the scaling of the input signal. Other blocks require that you specify the output scaling with the **Output scaling** parameter. Still other blocks provide you with the option of inheriting the output scaling (and data type) information from a driving block, or specifying the scaling.

  For the latter case, you control how the output scaling (and data type) is specified with the **Output data type and scaling** parameter list. This list supports three choices: `Specify via dialog`, `Inherit via internal rule`, and `Inherit via back propagation`. Note that some fixed-point blocks support only two of the three choices.

  If you select `Specify via dialog`, you must explicitly specify the output scaling with the **Output scaling** parameter. If you select `Inherit via internal rule`, the output scaling is inherited from the input(s). If you select `Inherit via back propagation`, the output scaling is inherited by back propagation; typically from the Data Type Propagation block. For information about the inheritance rules, refer to the description in “Selecting the Data Type” on page 10-4.

The supported scaling modes for generalized fixed-point data types are given below. Default scaling is used for all other fixed-point data types.

Table 10-2: Scaling Modes for Generalized Fixed-Point Data Types

Scaling mode	Description
Radix point-only	Specify radix point-only (powers-of-two) scaling. For example, a scaling of `2^-10` (or `pow2(-10)`) places the radix point at a location 10 places to the left of the least significant bit.
Slope/bias	Specify slope/bias scaling. For example, a scaling of `[5/9 10]` specifies a slope of 5/9 and a bias of 10. When using this mode, you must specify a positive slope.

Note that some blocks provide a form of radix point-only scaling for constant vectors and constant matrices. Refer to "Example: Constant Scaling for Best Precision" on page 3-12 for more information.

Locking the Output Scaling

If the **Lock output scaling so autoscaling tool can't change it** check box is checked, then the automatic scaling tool `autofixexp` will not change the **Output scaling** parameter value. Otherwise, the automatic scaling tool is free to adjust the scaling. You can run `autofixexp` directly from the command line, or through the Fixed-Point Blockset Interface tool, `fxptdlg`.

Rounding

You can choose the rounding mode for the block operation with the **Round toward** parameter list. The available rounding modes are shown below.

Table 10-3: Rounding Modes

Rounding Mode	Description
`Zero`	Round the output towards zero.
`Nearest`	Round the output towards the nearest representable number, with the exact midpoint rounded towards positive infinity.

Table 10-3: Rounding Modes (Continued)

Rounding Mode	Description
`Ceiling`	Round the output towards positive infinity.
`Floor`	Round the output towards negative infinity.

Handling Overflows

Overflow handling for fixed-point numbers is specified with the **Saturate to max or min when overflows occur** check box. If checked, fixed-point overflow results saturate. Otherwise, overflow results wrap. Whenever a result saturates, a warning is displayed.

Overriding with Doubles

If the **Override data type(s) with doubles** check box is checked, then the **Parameter data type** and **Output data type** parameter values are ignored. Instead, parameters and outputs are represented using double-precision floating-point numbers. Also, any calculations are performed using floating-point arithmetic.

An exception to this rule is when parameters or outputs contain a bias. In this case, the bias is not ignored in subsequent fixed-point operations.

If the parameter and output data types are both floating-point, the check box is not available.

Logging Simulation Results

The minimum and maximum values produced by the simulation are logged if the **Log minimums and maximums** check box is checked. The logged values are stored in the `FixPtSimRanges` global cell array in the MATLAB workspace. You can access these values with the `showfixptsimranges` script or with the Fixed-Point Blockset Interface tool, `fxptdlg`.

In addition to logging the minimum and maximum simulation values, overflow information is also logged. If an overflow occurs, then a warning, an error, or nothing occurs depending on how the **Data Overflow** parameter of Simulink's **Simulation Parameters** dialog box is configured.

Block Icon Labels

Many blockset icons look like those of built-in Simulink blocks. For this reason, all fixed-point icons have an “F” (for “Fixed-Point”) associated with them. An “F” in the lower right (upper left) corner of the icon means the block output (input) is a Fixed-Point Blockset data type.

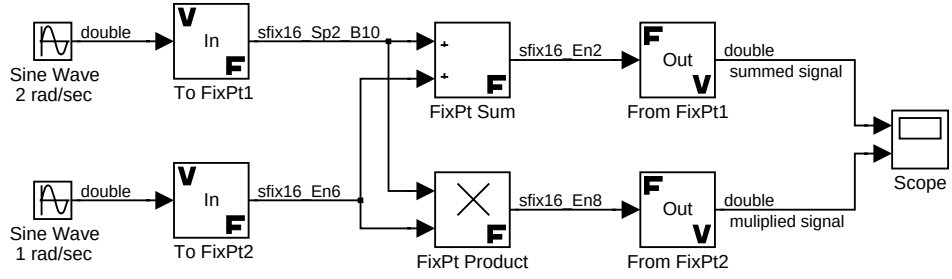
The Gateway In, Gateway In Inherited, and Gateway Out blocks have additional labels, which reflect how the input and output signals are treated. If the block input or output is treated as a real-world value, then a “V” appears by the relevant port. If the block input or output is treated as a stored integer, then an “I” appears by the relevant port.

Many blocks have additional labels that indicate logical operations, arithmetic operations, numerical values, and so on. These labels will help you to quickly understand the behavior of fixed-point models without examining individual block dialog boxes.

Port Data Type Display

To display the data types of ports in your model, select **Port Data Types** from Simulink's **Format** menu.

The port display for fixed-point signals consists of three parts: the data type, the number of bits, and the scaling. The data type and number of bits reflect the block’s **Output data type** parameter value or the data type that is inherited from the driving block. The scaling reflects the block’s **Output scaling** parameter value or the scaling that is inherited from the driving block. For example, the data type displays for the Fixed-Point Sine demo are shown below.



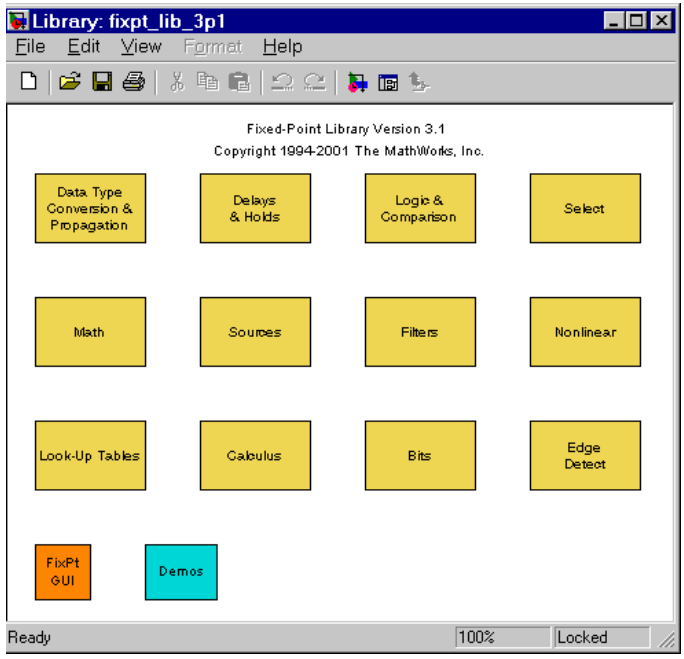
The data type display associated with the To FixPt1 block indicates that the output data type is `sfix(16)` (a signed 16-bit generalized fixed-point number) with slope/bias scaling of `[1/5, 10]`. Note that this scaling is not the block's default scaling. The data type display associated with the To FixPt2 block indicates that the output data type is `sfix(16)` with radix point-only scaling of `2^-6`.

The Fixed-Point Blockset Libraries

To open the main Fixed-Point library, at the MATLAB prompt type

```
fixpt
```

This opens the main library window as shown below.



The main library contains twelve sublibraries. To open a sublibrary, double-click on its icon. The following table describes how the Fixed-Point Blockset blocks are grouped into the sublibraries.

Bits	
Bit Clear	Set the specified bit of the stored integer to zero.

Bits	
Bit Set	Set the specified bit of the stored integer to one.
Bitwise Operator	Perform the specified bitwise operation on the inputs.

Calculus	
Accumulator	Compute a cumulative sum.
Accumulator Resettable	Compute a cumulative sum with external Boolean reset.
Accumulator Resettable Limited	Compute a limited cumulative sum with external Boolean reset.
Derivative	Compute a discrete time derivative.
Difference	Calculate the change in a signal over one time step.
Integrator Backward	Perform discrete-time integration of a signal using the backward method.
Integrator Backward Resettable	Perform discrete-time integration of a signal using the backward method, with external Boolean reset.
Integrator Backward Resettable Limited	Perform discrete-time limited integration of a signal using the backward method, with external Boolean reset.
Integrator Forward	Perform discrete-time integration of a signal using the forward method.
Integrator Forward Resettable	Perform discrete-time integration of a signal using the forward method, with external Boolean reset.

Calculus	
Integrator Forward Resettable Limited	Perform discrete-time limited integration of a signal using the forward method, with external Boolean reset.
Integrator Trapezoidal	Perform discrete-time integration of a signal using the trapezoidal method.
Integrator Trapezoidal Resettable	Perform discrete-time integration of a signal using the trapezoidal method, with external Boolean reset.
Integrator Trapezoidal Resettable Limited	Perform discrete-time limited integration of a signal using the trapezoidal method, with external Boolean reset.
Sample Time Multiply	Support calculations involving sample time.

Data Type	
Conversion	Convert from one Fixed-Point Blockset data type to another.
Conversion Inherited	Convert from one Fixed-Point Blockset data type to another, and inherit the data type and scaling.
Data Type Duplicate	Set all inputs to the same data type.
Data Type Propagation	Configure the data type and scaling of the propagated signal based on information from the reference signals.
Gateway In	Convert a Simulink data type to a Fixed-Point Blockset data type.

Data Type	
Gateway In Inherited	Convert a Simulink data type to a Fixed-Point Blockset data type, and inherit the data type and scaling.
Gateway Out	Convert a Fixed-Point Blockset data type to a Simulink data type.
Scaling Strip	Remove scaling and map to a built in integer.

Delays & Holds	
Integer Delay	Delay a signal N sample periods.
Tapped Delay	Delay a scalar signal multiple sample periods and output all the delayed versions.
Unit Delay	Delay a signal one sample period.
Unit Delay Enabled	Delay a signal one sample period.
Unit Delay Enabled External IC	Delay a signal one sample period.
Unit Delay Enabled Resettable	Delay a signal one sample period.
Unit Delay Enabled Resettable External IC	Delay a signal one sample period.
Unit Delay External IC	Delay a signal one sample period.
Unit Delay Resettable	Delay a signal one sample periodt.
Unit Delay Resettable External IC	Delay a signal one sample period.
Unit Delay With Preview Enabled	Support calculations that have feedback and depend on the current input.

Delays & Holds	
Unit Delay With Preview Enabled Resettable	Support calculations that have feedback and depend on the current input.
Unit Delay With Preview Enabled Resettable External RV	Support calculations that have feedback and depend on the current input.
Unit Delay With Preview Resettable	Support calculations that have feedback and depend on the current input.
Unit Delay With Preview Resettable External RV	Support calculations that have feedback and depend on the current input.
Zero-Order Hold	Implement a zero-order hold of one sample period.

Edge Detect	
Detect Change	Detect a change in a signal's value.
Detect Decrease	Detect a decrease in a signal's value.
Detect Fall Negative	Detect a falling edge when the signal's value decreases to a strictly negative value, and its previous value was nonnegative.
Detect Fall Nonpositive	Detect a falling edge when the signal's value decreases to a nonpositive value, and its previous value was strictly positive.
Detect Increase	Detect an increase in a signal's value.

Edge Detect	
Detect Rise Nonnegative	Detect a rising edge when a signal's value increases to a nonnegative value, and its previous value was strictly negative.
Detect Rise Positive	Detect a rising edge when a signal's value increases to a strictly positive value, and its previous value was nonpositive.

Filters	
Filter Direct Form I	Implement a Direct Form I realization of a filter.
Filter Direct Form I Time Varying	Implement a time varying Direct Form I realization of a filter.
Filter Direct Form II	Implement a Direct Form II realization of a filter.
Filter Direct Form II Time Varying	Implement a time varying Direct Form II realization of a filter.
Filter First Order	Implement a discrete-time first order filter.
Filter Lead or Lag	Implement a discrete-time lead or lag filter.
Filter Real Zero	Implement a discrete time filter that has a real zero and no pole.
FIR	Implement a fixed-point finite impulse response (FIR) filter.
State-Space	Implement discrete-time state space.

Logic & Comparison	
Compare to Constant	Determine if a signal is equal to the specified constant.
Compare To Zero	Determine if a signal is equal to zero.
Interval Test	Determine if a signal is in a specified interval.
Interval Test Dynamic	Determine if a signal is in a specified interval.
Logical Operator	Perform the specified logical operation on the inputs.
Relational Operator	Perform the specified relational operation on the inputs.

LookUp	
Cosine	Implement a cosine function in fixed-point using a lookup table approach that exploits quarter wave symmetry.
Look-Up Table	Approximate a one-dimensional function using a selected look-up method.
Look-Up Table Dynamic	Provide a region of zero output.
Look-Up Table (2-D)	Approximate a two-dimensional function using a selected look-up method.
Sine	Implement a sine function in fixed-point using a lookup table approach that exploits quarter wave symmetry.

Math Blocks	
Abs	Output the absolute value of the input.
Decrement Real World	Decrease the real world value of the signal by one
Decrement Stored Integer	Decrease the stored value of a signal by one
Decrement Time To Zero	Decrease the real world value of the signal by the sample time, but only to zero.
Decrement To Zero	Decrease the real world value of a signal by one, but only to zero.
Gain	Multiply the input by a constant.
Increment Real World	Increase the real world value of the signal by one.
Increment Stored Integer	Increase the stored integer value of a signal by one.
Matrix Gain	Multiply the input by a constant matrix.
MinMax	Determine the minimum or maximum input value.
MinMax Running Resettable	Determine the minimum or maximum of a signal over time.
Product	Multiply or divide inputs.
Sum	Add or subract inputs.
Unary Minus	Negate the input.

Nonlinear Blocks	
Dead Zone	Provide a region of zero output.
Dead Zone Dynamic	Set the input within the bounds to zero.
Rate Limiter	Limit the rising and falling rates of the signal.
Rate Limiter Dynamic	Limit the rising and falling rates of the signal.
Relay	Switch output between two constants.
Saturation	Bound the range of the input.
Saturation Dynamic	Bound the range of the input.
Sign	Indicate the sign of the input.
Wrap To Zero	Set output to zero if input is above threshold.

Select	
MultiPort Switch	Switch output between different inputs based on the value of the first input
Switch	Switch output between the first input and the third input based on the value of the second input

Sources	
Constant	Generate a constant value
Counter Free	Count up and overflow back to zero after the maximum value possible is reached for the specified number of bits
Counter Limited	Count up, and wrap back to zero after outputting the specified upper limit.
Repeating Sequence Interpolated	Output a discrete-time sequence and repeat, interpolating between data points
Repeating Sequence Stair	Output a discrete time sequence and repeat.

Abs

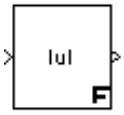
Purpose

Output the absolute value of the input

Library

Math

Description

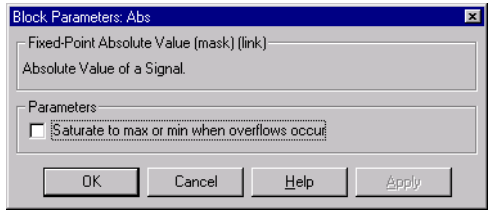


The Abs block is a masked S-function that outputs the absolute value of the input.

For signed data types, the absolute value of the most negative value is problematic since it is not representable by the data type. In this case, the behavior of the block is controlled by the **Saturate to max or min when overflows occur** check box. If checked, the absolute value of the data type saturates to the most positive value. If not checked, the absolute value of the most negative value has no effect.

For example, suppose the block input is an 8-bit signed integer. The range of this data type is from -128 to 127, and the absolute value of -128 is not representable. If the **Saturate to max or min when overflows occur** check box is checked, then the absolute value of -128 is 127. If it is not checked, then the absolute value of -128 remains at -128.

Parameters and Dialog Box



Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	N/A

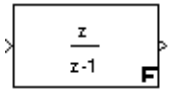
Purpose

Compute a cumulative sum

Library

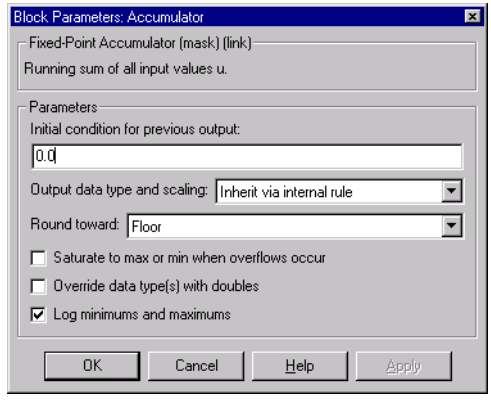
Calculus

Description



At time step n, the Accumulator block computes a cumulative sum of all input values u up to time n and outputs the sum.

Parameters and Dialog Box



Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same data type as the input
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

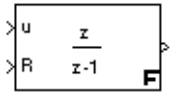
Purpose

Compute a cumulative sum with external Boolean reset

Library

Calculus

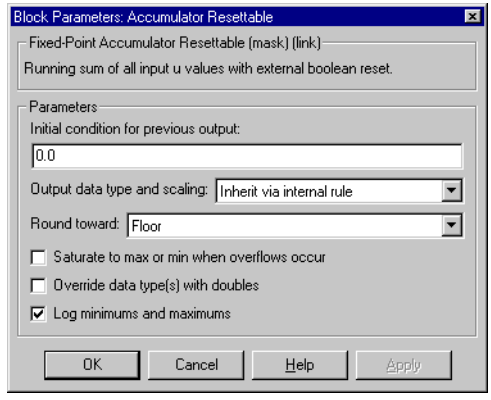
Description



The Accumulator Resettable block computes a cumulative sum, based on the values of an external Boolean reset signal.

The block can reset its state based on an external reset signal R. The block has two input ports, one for the input signal u, and another for the reset signal R. When the reset is false at time n, the block adds the current value of the input signal u to the sum at time n-1. When the reset is true at time n, the block resets the sum to the value of the **Initial condition for previous output** parameter, and outputs the sum.

Parameters and Dialog Box



Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Accumulator, and related blocks

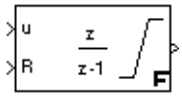
Purpose

Compute a limited cumulative sum with external Boolean reset

Library

Calculus

Description



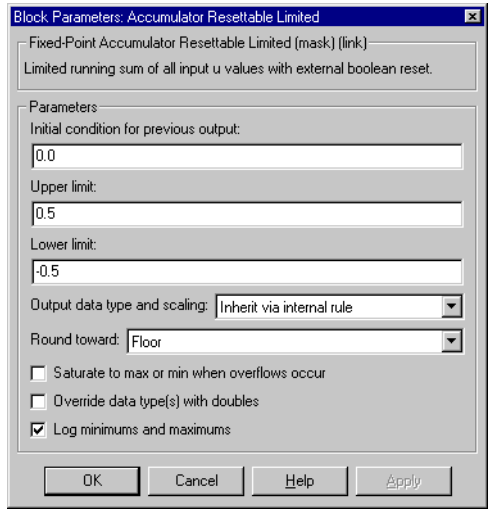
The Accumulator Resettable Limited block computes a cumulative sum, based on the values of an external Boolean reset signal.

The block can reset its state based on an external reset signal R. When the cumulative sum reaches one of the limits given by the **Upper limit** and **Lower limit** parameters, the sum saturates to that limit.

The block has two input ports, one for the input signal u, and another for the reset signal R. When the reset R is false at time n, the block adds the current value of the input signal u to the sum at time n-1. When the cumulative sum is outside the limits given by the **Upper limit** and **Lower limit** parameters, the sum saturates to one of the bounds.

When the reset R is true at time n, the block resets the sum to the value of the **Initial condition for previous output** parameter, and outputs the sum.

Parameters and Dialog Box



Initial condition for previous output

Set the initial condition for the previous output.

Upper limit

The upper limit for saturation of the cumulative sum.

Lower limit

The lower limit for saturation of the cumulative sum.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Accumulator, and related blocks

Purpose

Set the specified bit of the stored integer to zero

Library

Bits

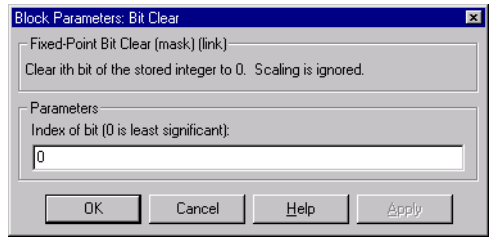
Description

The Bit Clear block is a masked block that sets the specified bit, given by its index, of the stored integer to zero. Scaling is ignored.

You can specify the bit to be set to zero with the **Index of bit** parameter, where bit zero is the least significant bit.

True floating-point data types are not supported.

Parameters and Dialog Box



Index of bit

Index of bit where bit 0 is the least significant bit.

Example

If the Bit Clear is turned on for bit 2, bit 2 is set to 0. A vector of constants `2.^[0 1 2 3 4]` is represented in binary as [00001 00010 00100 01000 10000]. With bit 2 set to 0, the result is [00001 00010 00000 01000 10000], which is represented in decimal as `[1 2 0 8 16]`.

Characteristics

Input Port	Any data type supported by the blockset, except a true floating-point data type
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

Bit Set

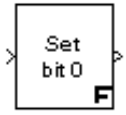
Purpose

Set the specified bit of the stored integer to one

Library

Bits

Description

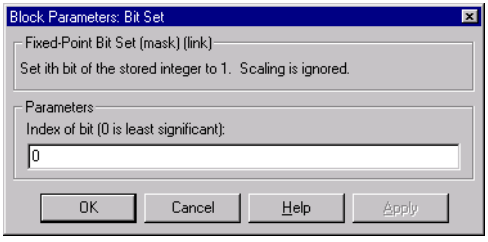


The Bit Set block is a masked block that sets the specified bit of the stored integer to one. Scaling is ignored.

You can specify the bit to be set to one with the **Index of bit** parameter, where bit zero is the least significant bit.

True floating-point data types are not supported.

Parameters and Dialog Box



Index of bit

Index of bit where bit 0 is the least significant bit.

Example

If the Bit Set is turned on for bit 2, bit 2 is set to 1. A vector of constants `2.^[0 1 2 3 4]` is represented in binary as [00001 00010 00100 01000 10000]. With bit 2 set to 1, the result is [00101 00110 00100 01100 10100], which is represented in decimal as `[5 6 4 12 20]`.

Characteristics

Input Port	Any data type supported by the blockset, except a true floating-point data type
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Bit Clear

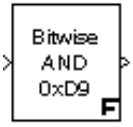
Purpose

Perform the specified bitwise operation on the inputs

Library

Bits

Description



The Bitwise Operator block is a masked S-function that performs the specified bitwise operation on its operands.

Unlike the logic operations performed by the Logical Operator block, bitwise operations treat the operands as a vector of bits rather than a single number. You select the bitwise Boolean operation with the **Operator** parameter list. The supported operations are given below.

Operation	Description
AND	TRUE if the corresponding bits are all TRUE
OR	TRUE if at least one of the corresponding bits is TRUE
NAND	TRUE if at least one of the corresponding bits is FALSE
NOR	TRUE if no corresponding bits are TRUE
XOR	TRUE if an odd number of corresponding bits are TRUE
NOT	TRUE if the input is FALSE (available only for single input)

Unlike Simulink's Bitwise Logical Operator block, the Bitwise Operator block does not support shift operations. Refer to "Shifts" on page 4-41 to learn how to perform shift operations with the Fixed-Point Blockset.

The size of the output depends on the number of inputs, their vector size, and the selected operator:

- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the bitwise logical complements of the input vector elements.
- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. If a bit mask is not specified, then the output is a scalar. If a bit mask is specified, then the output is a vector.

- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the operation is performed between corresponding elements of the vectors to produce a vector output.

When configured as a multi-input XOR gate, this block performs an addition-modulo-two operation as mandated by the IEEE Standard for Logic Elements.

If the **Use bit mask** check box is not checked, then the block can accept multiple inputs. You select the number of input ports with the **Number of input ports** parameter. The input data types must be identical.

If the **Use bit mask** check box is checked, then a single input is associated with the bit mask you specify with the **Bit mask** parameter. You specify the bit mask using any valid MATLAB expression. For example, you can specify the bit mask 00100101 as `2^5+2^2+2^0`. Alternatively, you can use strings to specify a hexadecimal bit mask such as `{'FE73','12AC'}`. If the bit mask is larger than the input signal data type, then it is ignored.

Note The output data type, which is inherited from the driving block, should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

The **Treat mask as** parameter list controls how the mask is treated. The possible values are `Real World Value` and `Stored Integer`. In terms of the general encoding scheme described in "Scaling" on page 3-5, `Real World Value` treats the mask as $V = SQ + B$ where S is the slope and B is the bias. `Stored Integer` treats the mask as a stored integer, Q. For more information about this parameter list, refer to the Gateway In block.

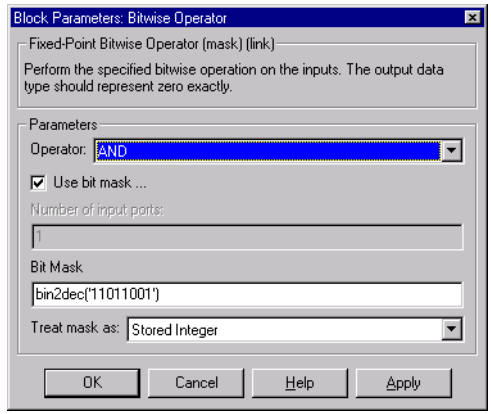
Remarks

You can use the bit mask to perform a bit set or a bit clear on the input. To perform a bit set, you configure the **Operator** parameter list to `OR` and create a bit mask with a 1 for each corresponding input bit that you want to set to 1. To perform a bit clear, you configure the **Operator** parameter list to `AND` and create a bit mask with a 0 for each corresponding input bit that you want to set to 0.

For example, suppose you want to perform a bit set on the fourth bit of an 8-bit input vector. The bit mask would be 00010000, which you can specify as `2^4` in

the **Bit mask** parameter. To perform a bit clear, the bit mask would be 11101111, which you can specify as `2^7+2^6+2^5+2^3+2^2+2^1+2^0` in the **Bit mask** parameter.

Parameters and Dialog Box



Operator

The bitwise logical operator associated with the specified operands.

Use bit mask

Specify if the bit mask is used (single input only).

Number of input ports

The number of inputs.

Bit mask

The bit mask to associate with a single input.

Treat mask as

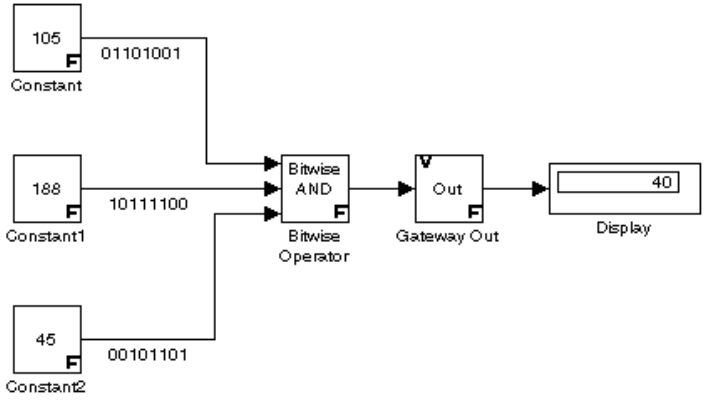
Treat the mask as a real-world value or as an integer.

Conversions

The **Bit mask** parameter is converted from a double to the input data type offline using round-to-nearest and saturation. Refer to “Parameter Conversions” on page 4-27 for more information about parameter conversions.

Example

To help you understand the Bitwise Operator block logic operations, consider the fixed-point model shown below.



The Constant blocks are configured to output an 8-bit unsigned integer (`uint(8)`). The results for all logic operations are shown below.

Operation	Binary Value	Decimal Value
AND	00101000	40
OR	11111101	253
NAND	11010111	215
NOR	00000010	2
XOR	11111000	248
NOT	N/A	N/A

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	No
Scalar Expansion	Of inputs

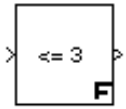
Purpose

Determine if a signal is equal to the specified constant

Library

Logic & Comparison

Description

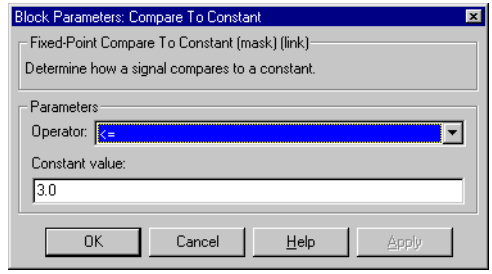


The Compare To Constant block is a masked block that determines if a signal is equal to the specified constant where:

- The output is true (not 0) when the input signal is equal to the specified constant.
- The output is false (equal to 0) when the input signal is not equal to the specified constant.

You enter the constant with the **Constant value** parameter.

Parameters and Dialog Box



Operator

Specify how the input is compared to the constant value.

Constant value

Specify the constant value that the input is compared with.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes

Compare To Zero

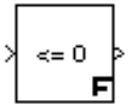
Purpose

Determine if a signal is equal to zero

Library

Logic & Comparison

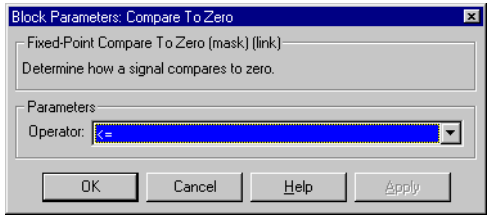
Description



The Compare To Zero block is a masked block that determines if a signal is equal to zero where:

- The output is true (not 0) when the input signal is equal to zero.
- The output is false (equal to 0) when the input signal is not equal to zero.

Parameters and Dialog Box



Operator

Specify how the input is compared to zero.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	N/A

See Also

Compare To Constant

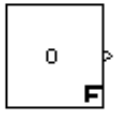
Purpose

Generate a constant value

Library

Sources

Description



The Constant block is a masked S-function that generates a constant value.

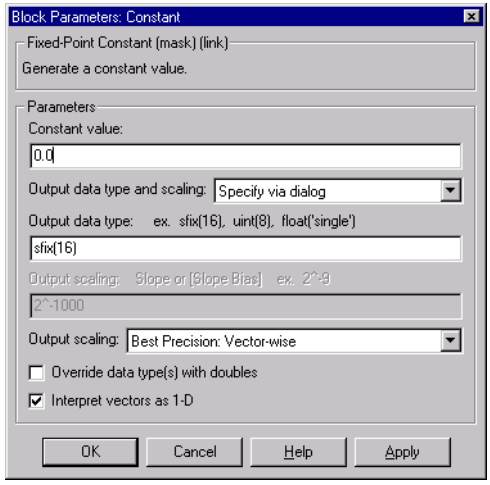
You specify constants with the **Constant value** parameter. A constant can be a scalar or a vector.

You specify the output scaling with the **Output scaling** parameter. Note that there are two dialog box parameters that control the output scaling: one associated with an edit field, and one associated with a parameter list. If **Output data type** is a generalized fixed-point number such as `sfix(16)`, the **Output scaling** parameter list provides you with these scaling modes:

- `Use Specified Scaling` – This mode uses the slope/bias or radix point-only scaling specified for the editable **Output scaling** parameter (for example, `2^-10`).
- `Best Precision: Vector-wise` – This mode produces a common radix point for each element of the **Constant value** vector based on the best precision for the largest value of the vector.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

Parameters and Dialog Box



Constant value

Constant value output by the block. It can be a scalar or vector. All members of the output vector must be the same data type.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling via back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. Additionally, if **Constant value** is specified as a vector, it can be scaled using the constant vector scaling modes for maximizing precision. These scaling modes are available only for generalized fixed-point data types.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Interpret vectors as 1-D

If checked, a column or row matrix value for the **Constant value** parameter results in a vector output whose elements are the elements of the row or column.

Conversions

The **Constant value** parameter is converted from a double to the specified output data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

Characteristics

Output Port	Any data type supported by the blockset
Direct Feedthrough	No
Scalar Expansion	No – the output is always the same size as **Constant value**

Conversion

Purpose

Convert from one Fixed-Point Blockset data type to another

Library

Data Type

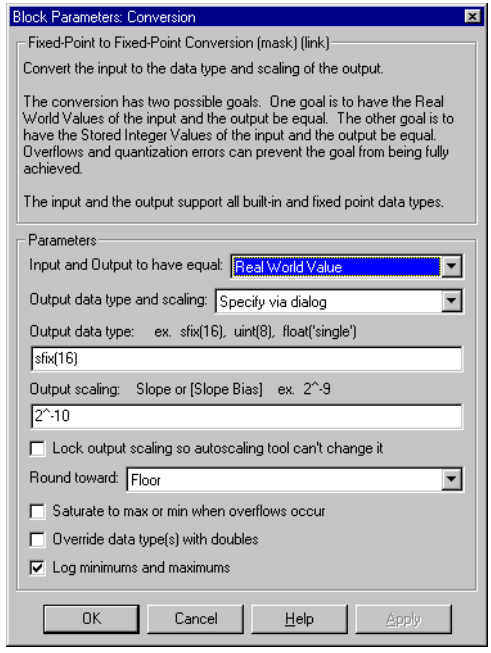
Description

The Conversion block is a masked S-function that converts from one Fixed-Point Blockset data type to another.

This block requires that you specify the data type and scaling for the conversion. If you want to inherit this information from an input signal, you should use the Conversion Inherited block.

For a detailed description of all block parameters, refer to “Block Parameters” on page 10-4. For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-27.

Parameters and Dialog Box



Input and Output to have equal

Specify the type of value of the input and output that are to be equal.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling via back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	N/A

See Also

Conversion Inherited

Conversion Inherited

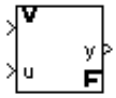
Purpose

Convert from one Fixed-Point Blockset data type to another, and inherit the data type and scaling

Library

Data Type

Description



The Conversion Inherited block is a masked S-function that forces dissimilar data types to be the same. The first (top) input is used as the reference signal and the second (bottom) input is converted to the reference type by inheriting the data type and scaling information. Either input will be scalar expanded such that the output has the same width as the widest input.

If you want to specify the data type and scaling when converting from one Fixed-Point Blockset data type to another, you should use the Conversion block.

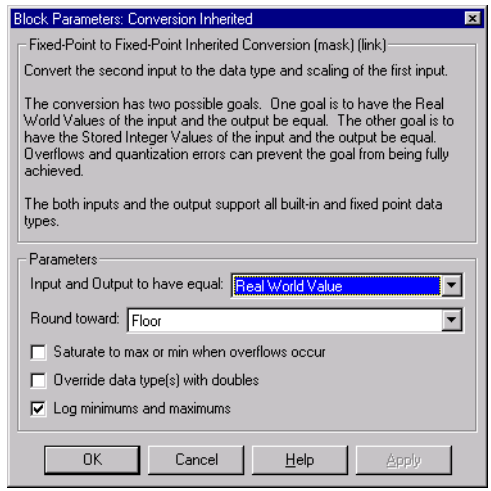
For a detailed description of all block parameters, refer to “Block Parameters” on page 10-4. For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-27.

Remarks

Inheriting the data type and scaling provides these advantages:

- It makes reusing existing models easier.
- It allows you to create new fixed-point models with less effort since you can avoid the detail of specifying the associated parameters.

Parameters and Dialog Box



Input and Output to have equal

Specify the type of value of the input and output that are to be equal.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the inherited data type is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes

See Also

Conversion

Cosine

Purpose

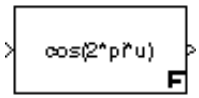
Implement a cosine function in fixed-point using a lookup table approach that exploits quarter wave symmetry

Library

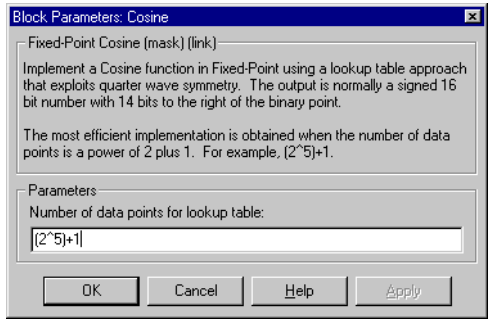
LookUp

Description

The Cosine block implements a cosine function using a lookup table that exploits quarter wave symmetry. The output is normally a signed 16 bit number with 14 bits to the right of the binary point.



Parameters and Dialog Box



Number of data points for lookup table

The number of data points in the lookup table

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	N/A

Purpose

Count up, and overflow back to zero after the maximum value possible is reached for the specified number of bits

Library

Sources

Description

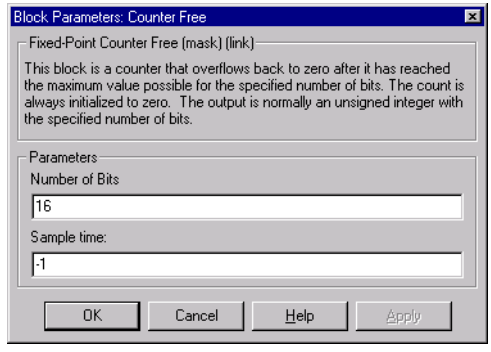
The Free Counter block is a masked block that counts up until the maximum possible value, $2^{Nbits} - 1$, is reached, where Nbits is the number of bits. Then the counter overflows to zero, and restarts counting up. The counter is always initialized to zero.

You can specify the number of bits with the **Number of Bits** parameter.

You can specify the sample time with the **Sample time** parameter.

The output is an unsigned integer. If the global doubles override is selected, the Free Counter does not wrap back to zero.

Parameters and Dialog Box



Number of Bits

Specified number of bits.

Sample time

Sample time.

Characteristics

Output Port	Unscaled integer or a floating-point data type
Scalar Expansion	No
Vectorized	No

Counter Limited

Purpose

Count up, and wrap back to zero after outputting the specified upper limit

Library

Sources

Description

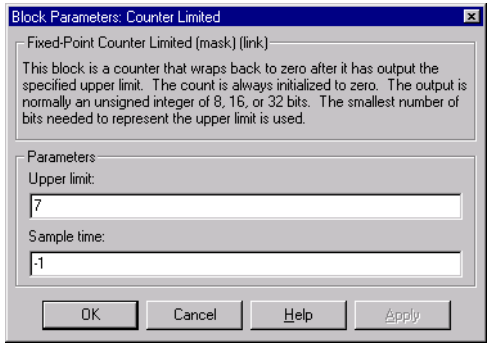
The Limited Counter block is a masked block that counts up until the specified upper limit is reached. Then the counter wraps back to zero, and restarts counting up. The counter is always initialized to zero.

You can specify the upper limit with the **Upper limit** parameter.

You can specify the sample time with the **Sample time** parameter. A **Sample time** of -1 means that the sample time is inherited.

The output is an unsigned integer of 8, 16, or 32 bits, with the smallest number of bits needed to represent the upper limit.

Parameters and Dialog Box



Upper limit

Upper limit.

Sample time

Sample time.

Characteristics

Output Port	Unscaled integer or a floating-point data type
Scalar Expansion	No
Vectorized	No

See Also

Counter Free

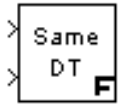
Purpose

Set all inputs to the same data type

Library

Data Type

Description

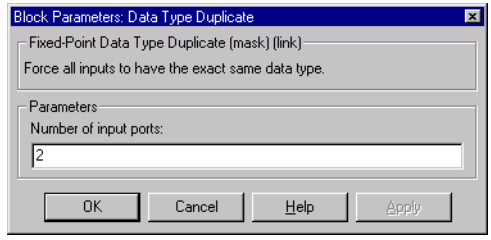


The Data Type Duplicate block is a masked S-function that forces all inputs to have exactly the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are completely independent.

You can use the Data Type Duplicate block to check for consistency of data types among blocks, where if all signals do not have the same data type, the block returns an error message.

You set the number of input ports with the **Number of input ports** parameter.

Parameters and Dialog Box



Number of input ports

Number of input ports.

Characteristics

Input Port	Any data type supported by the blockset
Scalar Expansion	Yes
States	0
Vectorized	Yes

Remarks

The Data Type Duplicate block is typically used such that one signal to the block controls the data type for all other blocks. The other blocks are set to inherit their data types via back propagation.

The block is also used in a user created library. These library blocks can be placed in any model, and the data type for all library blocks are configured

according to the usage in the model. To create a library block with more complex data type rules than duplication, use the Data Type Propagation block.

Purpose

Configure the data type and scaling of the propagated signal based on information from the reference signals

Library

Data Type

Description

The Data Type Propagation block allows you to control the data type and scaling of signals in your model. You can use this block in conjunction with fixed-point blocks that have their **Specify data type and scaling** parameter configured to `Inherit via back propagation`.

The block has three inputs: Ref1 and Ref2 are the reference inputs, while the Prop input back propagates the data type and scaling information gathered from the reference inputs. This information is then passed on to other fixed-point blocks.

The block provides you with many choices for propagating data type and scaling information. For example, you can:

- Use the number of bits from the Ref1 reference signal, or use the number of bits from widest reference signal.
- Use the range from the Ref2 reference signal, or use the range of the reference signal with the greatest range.
- Use a bias of zero, regardless of the biases used by the reference signals.
- Use the precision of the reference signal with the least precision.

You specify how data type information is propagated with the **Propagated data type** parameter list. If the parameter list is configured as `Specify via dialog`, then you manually specify the data type via the **Propagated data type** edit field. Refer to “Selecting the Data Type” on page 10-4 to learn how to specify the data type. If the parameter list is configured as `Inherit via propagation rule`, then you must use the parameters described in “Inheriting Data Type Information” on page 10-52.

You specify how scaling information is propagated with the **Propagated scaling** parameter list. If the parameter list is configured as `Specify via dialog`, then you manually specify the scaling via the **Propagated scaling** edit field. Refer to “Selecting the Scaling” on page 10-6 to learn how to specify the scaling. If the parameter list is configured as `Inherit via propagation rule`, then you must use the parameters described in “Inheriting Scaling Information” on page 10-54.

Remarks

After you use the information from the reference signals, you can apply a second level of adjustments to the data type and scaling by using individual multiplicative and additive adjustments. This flexibility has a variety of uses. For example, if you are targeting a DSP, then you can configure the block so that the number of bits associated with a MAC (multiply and accumulate) operation is twice as wide as the input signal, and has a certain number of guard bits added to it.

The Data Type Propagation block also provides a mechanism to force the computed number of bits to a useful value. For example, if you are targeting a 16-bit micro, then the target C compiler is likely to support sizes of only 8 bits, 16 bits, and 32 bits. The block will force these three choices to be used. For example, suppose the block computes a data type size of 24 bits. Since 24 bits is not directly usable by the target chip, the signal is forced up to 32 bits, which is natively supported.

There is also a method for dealing with floating-point reference signals. This makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or visa versa.

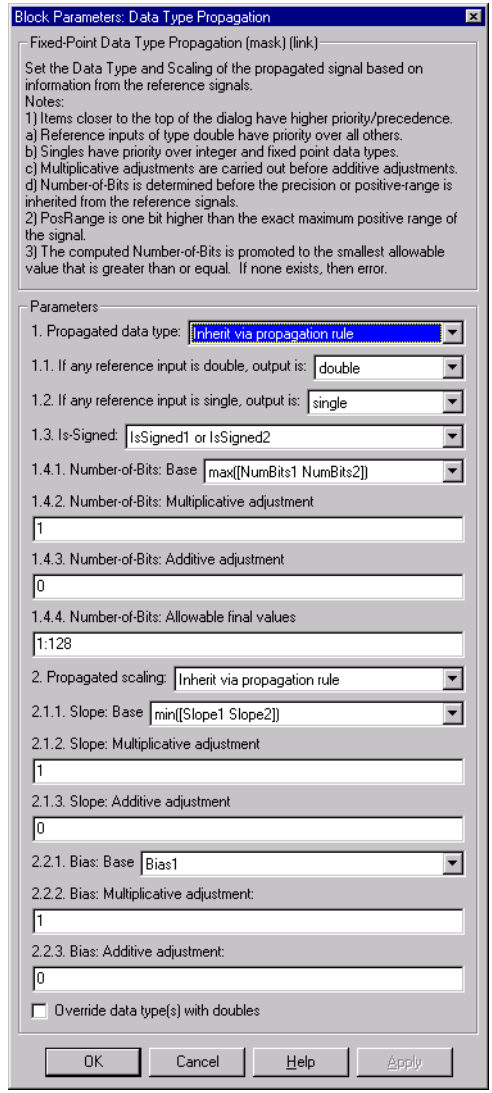
The Data Type Propagation block allows you to set up libraries of useful subsystems that will be properly configured based on the connected signals. Without this data type propagation process, a subsystem that you use from a library will almost certainly not work as desired with most integer or fixed-point signals, and manual intervention to configure the data type and scaling would be required. This block can eliminate the manual intervention in many situations.

Precedence Rules

The precedence of the dialog box parameters decreases from top to bottom. Additionally:

- Double-precision reference inputs have precedence over all other data types.
- Single-precision reference inputs have precedence over integer and fixed-point data types.
- Multiplicative adjustments are carried out before additive adjustments.
- The number of bits is determined before the precision or positive range is inherited from the reference inputs.

Parameters and Dialog Box



Propagated data type

Use the parameter list to propagate the data type via the dialog box, or inherit the data type from the reference signals. Use the edit field to specify the data type via the dialog box.

Propagated scaling

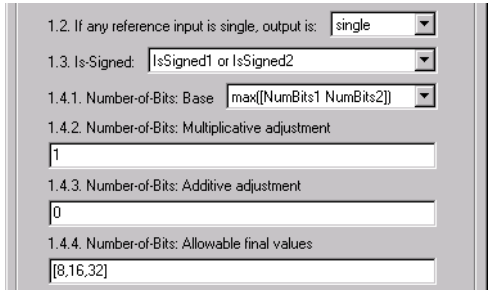
Use the parameter list to propagate the scaling via the dialog box, or inherit the scaling from the reference signals. Use the edit field to specify the scaling via the dialog box.

Override data type(s) with doubles

If checked, the data type is overridden with doubles.

Inheriting Data Type Information

If the **Propagated data type** parameter is `Inherit via propagation rule`, then these dialog box parameters are available to you.



The **If any reference input is single, output is** parameter list can be `single` or `double`. This parameter makes it easier to create designs that are easily retargeted from fixed-point chips to floating-point chips or visa versa.

The **Is-Signed** parameter list specifies the sign of Prop. The parameter values are described below.

Parameter Value	Description
`IsSigned1`	Prop is a signed data type if Ref1 is a signed data type.
`IsSigned2`	Prop is a signed data type if Ref2 is a signed data type.

Parameter Value	Description
`IsSigned1 or IsSigned2`	Prop is a signed data type if either Ref1 or Ref2 are signed data types.
`TRUE`	Ref1 and Ref2 are ignored, and Prop is always a signed data type.
`FALSE`	Ref1 and Ref2 are ignored, and Prop is always an unsigned data type.

For example, if the Ref1 signal is `ufix(16)`, the Ref2 signal is `sfix(16)`, and the **Is-Signed** parameter is `IsSigned1 or IsSigned2`, then Prop is forced to be a signed data type.

The **Number-of-bits: base** parameter list specifies the number of bits used by Prop for the base data type. The parameter values are described below.

Parameter Value	Description
`NumBits1`	The number of bits for Prop is given by the number of bits for Ref1.
`NumBits2`	The number of bits for Prop is given by the number of bits for Ref2.
`max([NumBits1 NumBits2])`	The number of bits for Prop is given by the reference signal with largest number of bits.
`min([NumBits1 NumBits2])`	The number of bits for Prop is given by the reference signal with smallest number of bits.
`NumBits1+NumBits2`	The number of bits for Prop is given by the sum of the reference signal bits.

Refer to "Targeting an Embedded Processor" in Chapter 7 for more information about the base data type.

The **Number-of-bits: Multiplicative adjustment** parameter allows you to adjust the number of bits used by Prop by including a multiplicative

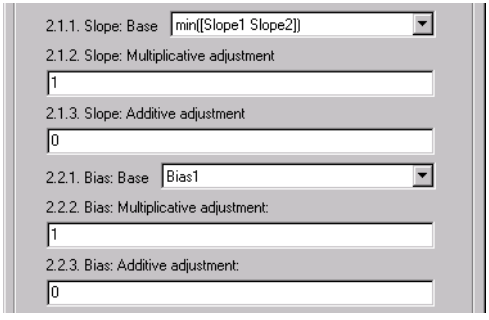
adjustment. For example, suppose you want to guarantee that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal. To do this, you configure this parameter to the value `2`.

The **Number-of-bits: Additive adjustment** parameter allows you to adjust the number of bits used by Prop by including an additive adjustment. For example, if you are performing multiple additions during a MAC operation, the result may overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value `4`.

The **Number-of-bits: Allowable final values** parameter allows you to force the computed number of bits used by Prop to a useful value. For example, if you are targeting a processor that supports only 8, 16, and 32 bits, then you configure this parameter to `[8,16,32]`. The block always propagates the smallest specified value that fits. If you want to allow all fixed-point data types, you would specify the value `1:128`.

Inheriting Scaling Information

If the **Propagated scaling** parameter is `Inherit via propagation rule`, then these dialog box parameters are available to you.



The **Slope: Base** parameter list specifies the slope used by Prop for the base data type. The parameter values are described below.

Parameter Value	Description
`Slope1`	The slope of Prop is given by the slope of Ref1.
`Slope2`	The slope of Prop is given by the slope of Ref2.

Parameter Value	Description
`max([Slope1 Slope2])`	The slope of Prop is given by the maximum slope of the reference signals.
`min([Slope1 Slope2])`	The slope of Prop is given by the minimum slope of the reference signals.
`Slope1*Slope2`	The slope of Prop is given by the product of the reference signal slopes.
`Slope1/Slope2`	The slope of Prop is given by the ratio of the Ref1 slope to the Ref2 slope.
`PosRange1`	The range of Prop is given by the range of Ref1.
`PosRange2`	The range of Prop is given by the range of Ref2.
`max([PosRange1 PosRange2])`	The range of Prop is given by the maximum range of the reference signals.
`min([PosRange1 PosRange2])`	The range of Prop is given by the minimum range of the reference signals.
`PosRange1*PosRange2`	The range of Prop is given by the product of the reference signal ranges.
`PosRange1/PosRange2`	The range of Prop is given by the ratio of the Ref1 range to the Ref2 range.

You control the precision of Prop with `Slope1` and `Slope2`, and you control the range of Prop with `PosRange1` and `PosRange2`. Additionally, `PosRange1` and `PosRange2` are one bit higher than the maximum positive range of the associated reference signal.

The **Slope: Multiplicative adjustment** parameter allows you to adjust the slope used by Prop by including a multiplicative adjustment. For example, if you want 3 bits of additional precision (with a corresponding decrease in range), the multiplicative adjustment is `2^-3`.

The **Slope: Additive adjustment** parameter allows you to adjust the slope used by Prop by including an additive adjustment. An additive slope adjustment is often not needed. The most likely use is to set the multiplicative

adjustment to 0, and set the additive adjustment to force the final slope to a specified value.

The **Bias: Base** parameter list specifies the bias used by Prop for the base data type. The parameter values are described below.

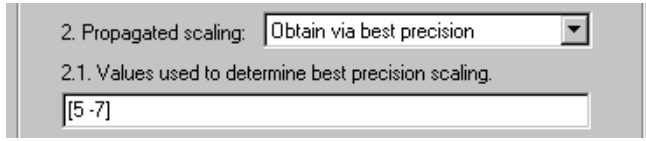
Parameter Value	Description
`Bias1`	The bias of Prop is given by the bias of Ref1.
`Bias2`	The bias of Prop is given by the bias of Ref2.
`max([Bias1 Bias2])`	The bias of Prop is given by the maximum bias of the reference signals.
`min([Bias1 Bias2])`	The bias of Prop is given by the minimum bias of the reference signals.
`Bias1*Bias2`	The bias of Prop is given by the product of the reference signal biases.
`Bias1/Bias2`	The bias of Prop is given by the ratio of the Ref1 bias to the Ref2 bias.
`Bias1+Bias2`	The bias of Prop is given by the sum of the reference biases.
`Bias1-Bias2`	The bias of Prop is given by the difference of the reference biases.

The **Bias: Multiplicative adjustment** parameter allows you to adjust the bias used by Prop by including a multiplicative adjustment.

The **Bias: Additive adjustment** parameter allows you to adjust the bias used by Prop by including an additive adjustment.

If you want to guarantee that the bias associated with Prop is zero, you should configure both the multiplicative adjustment and the additive adjustment to 0.

If the **Propagated scaling** parameter is `Obtain via best precision`, then the following dialog box parameters are available to you.



You specify any values, such as the upper and lower limits on the propagated input, for the **Values used to determine best precision scaling**, which constrains the precision chosen to apply to those limits. Based on the data type, the scaling will automatically be selected such that these values can be represent with no overflow error and minimum quantization error.

Characteristics

Input Ports	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

Dead Zone

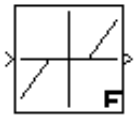
Purpose

Provide a region of zero output

Library

Nonlinear

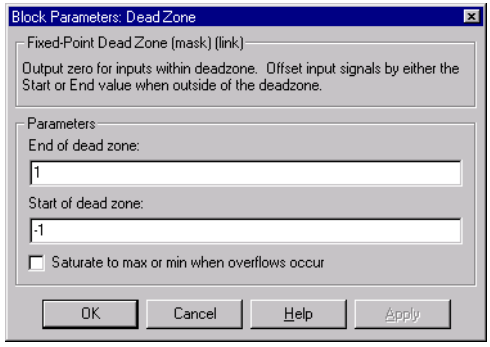
Description



The Dead Zone block is a masked S-function that generates zero output within a specified region, called its dead zone. The lower limit of the dead zone is specified with the **Start of dead zone** parameter, while the upper limit of the dead zone is specified with the **End of dead zone** parameter. The block output depends on the input and dead zone:

- If the input is within the dead zone (greater than the lower limit and less than the upper limit), the output is zero.
- If the input is greater than or equal to the upper limit, the output is the input minus the upper limit.
- If the input is less than or equal to the lower limit, the output is the input minus the lower limit.

Parameters and Dialog Box



Start of dead zone

The lower limit of the dead zone.

End of dead zone

The upper limit of the dead zone.

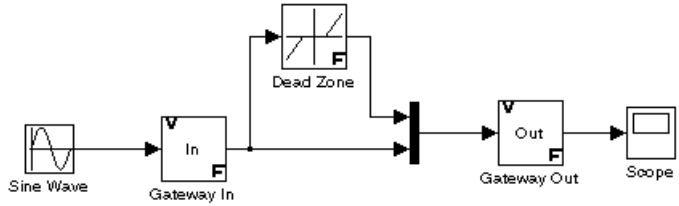
Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

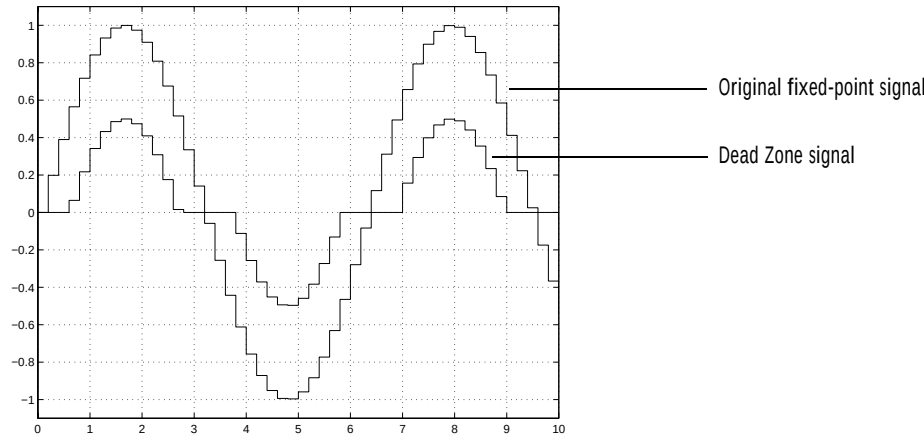
Example

Consider the model shown below, which compares a fixed-point signal and the output generated by the Dead Zone block. The signal source is a sine wave with unit amplitude.

The **Start of dead zone** parameter is configured to `-0.5` and the **End of dead zone** parameter is configured to `0.5`.



The resulting output is shown below.



Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes, of parameters

Dead Zone Dynamic

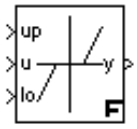
Purpose

Set inputs within the bounds to zero

Library

Nonlinear

Description

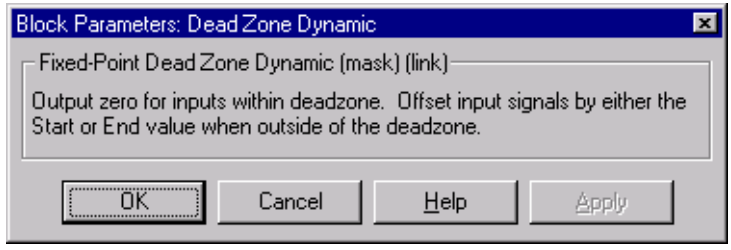


The Dead Zone Dynamic block is a masked block that dynamically bounds the range of the input signal, providing a region of zero output. The bounds change according to the upper and lower limit input signals where:

- The input within the bounds is set to zero.
- The input below the lower limit is shifted down by the lower limit.
- The input above the upper limit is shifted down by the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Dead Zone

Purpose

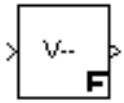
Decrease the real world value of the signal by one

Library

Math

Description

The Decrement Real World block is a masked block that decreases the real world value of the signal by one. Overflows always wrap.



Parameters and Dialog Box

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

Decrement Stored Integer

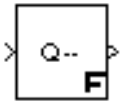
Purpose

Decrease the stored integer value of a signal by one

Library

Math

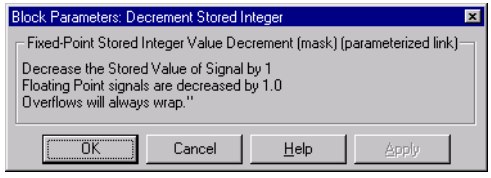
Description



The Decrement Stored Integer block is a masked block that decreases the stored integer value of a signal by one.

Floating-point signals are also decreased by one, and overflows always wrap.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Decrement Real World, and related blocks

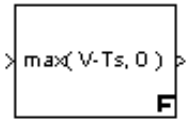
Purpose

Decrease the real world value of the signal by the sample time, but only to zero.

Library

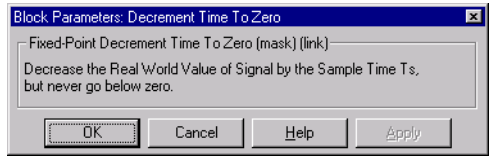
Math

Description



The Decrement Time To Zero block is a masked block that decreases the real world value of the signal by the sample time, Ts. The output will never go below zero.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Decrement Real World, and related blocks

Decrement To Zero

Purpose

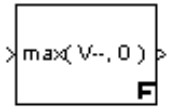
Decreases the real world value of a signal by one, but only to zero.

Library

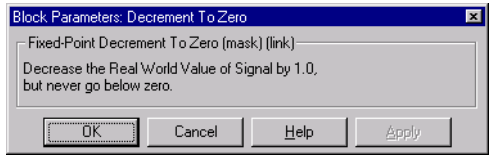
Math

Description

The Decrement To Zero block is a masked block that decreases the real world value of the signal by one. The output will never go below zero.



Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Decrement Real World, and related blocks

Purpose

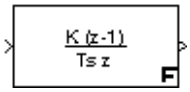
Compute a discrete time derivative

Library

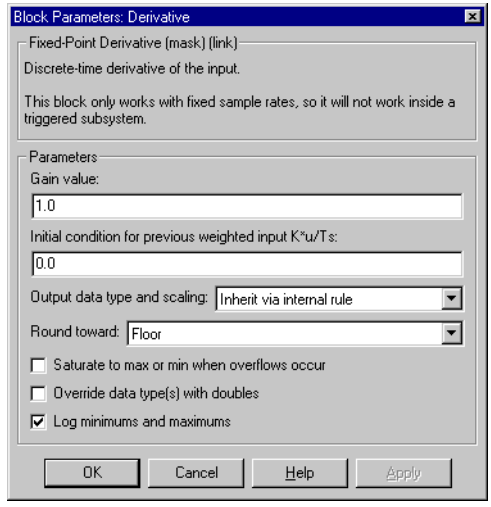
Calculus

Description

The Derivative block computes a discrete time derivative, by subtracting the input value at the previous time step from the current value, and dividing by the sample time.



Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous weighted input K*u/Ts

Set the initial condition for the previous scaled input.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

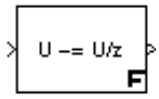
Purpose

Detect a change in a signal's value

Library

Edge Detect

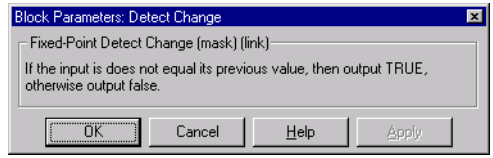
Description



The Detect Change block is a masked block that determines if an input does not equal its previous value where:

- The output is true (not 0), when the input signal does not equal its previous value.
- The output is false (equal to 0), when the input signal equals its previous value.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

Detect Decrease

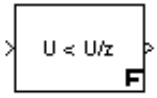
Purpose

Detect a decrease in a signal's value

Library

Edge Detect

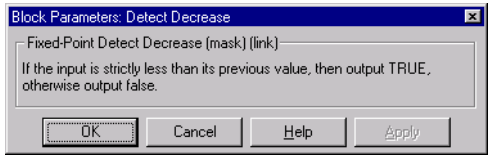
Description



The Detect Decrease block is a masked block that determines if an input is strictly less than its previous value where:

- The output is true (not 0), when the input signal is less than its previous value.
- The output is false (equal to 0), when the input signal is greater than or equal to its previous value.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

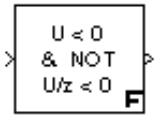
Purpose

Detect a falling edge when the signal's value decreases to a strictly negative value, and its previous value was nonnegative

Library

Edge Detect

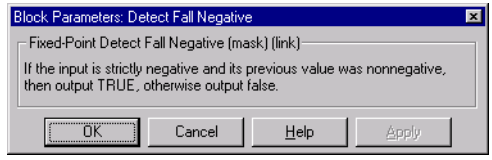
Description



The Detect Fall Negative block is a masked block that determines if the input is less than zero, and its previous value was greater than or equal to zero where:

- The output is true (not 0), when the input signal is less than zero, and its previous value was greater than or equal to zero.
- The output is false (equal to 0), when the input signal is greater than or equal to zero, or if the input signal is nonnegative, its previous value was positive or zero.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

Detect Fall Nonpositive

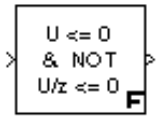
Purpose

Detect a falling edge when the signal's value decreases to a nonpositive value, and its previous value was strictly positive

Library

Edge Detect

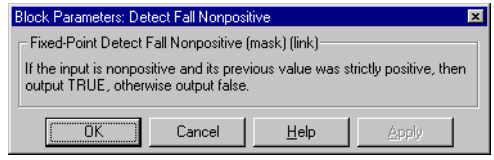
Description



The Detect Fall Nonpositive block is a masked block that determines if the input is less than or equal to zero, and its previous value was positive where:

- The output is true (not 0), when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0), when the input signal is greater than zero, or if it is nonpositive, its previous value was nonpositive.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

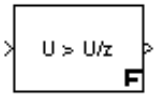
Purpose

Detect an increase in a signal's value

Library

Edge Detect

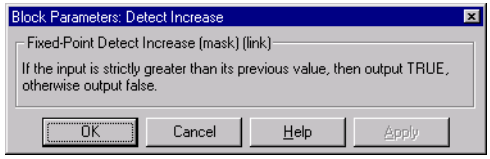
Description



The Detect Increase block is a masked block that determines if an input is strictly greater than its previous value where:

- The output is true (not 0), when the input signal is greater than its previous value.
- The output is false (equal to 0), when the input signal is less than or equal to its previous value.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

Detect Rise Nonnegative

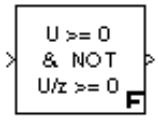
Purpose

Detect a rising edge when a signal's value increases to a nonnegative value, and its previous value was strictly negative

Library

Edge Detect

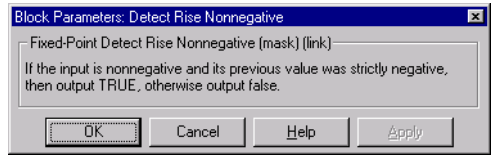
Description



The Detect Rise Nonnegative block is a masked block that determines if the input is greater than or equal to zero, and its previous value was less than zero where:

- The output is true (not 0), when the input signal is greater than or equal to zero, and its previous value was less than zero.
- The output is false (equal to 0), when the input signal is less than zero, or if nonnegative, its previous value was greater than or equal to zero.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

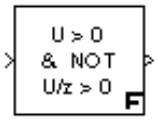
Purpose

Detect a rising edge when a signal's value increases to a strictly positive value, and its previous value was nonpositive

Library

Edge Detect

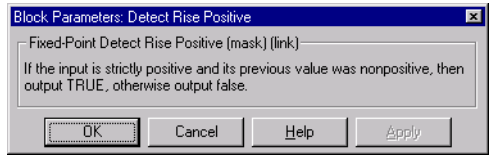
Description



The Detect Rise Positive block is a masked block that determines if the input is strictly positive, and its previous value was nonpositive where:

- The output is true (not 0), when the input signal is greater than zero, and its previous value was less than zero.
- The output is false (equal to 0), when the input is negative or zero, or if the input is positive, its previous value was also positive.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	An 8-bit unsigned integer
Direct Feedthrough	Yes
Scalar Expansion	Yes
Vectorized	Yes

See Also

Detect Change, and related blocks

Difference

Purpose

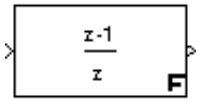
Calculate the change in a signal over one time step

Library

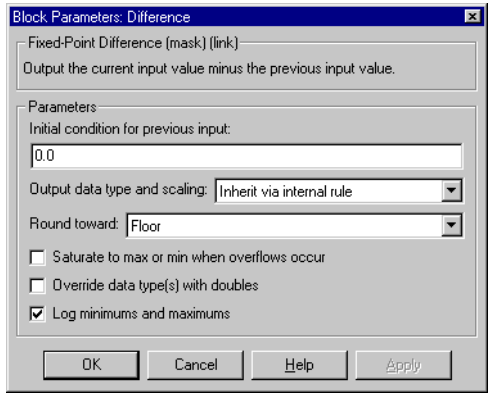
Calculus

Description

The Difference block outputs the current input value minus the previous input value.



Parameters and Dialog Box



Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

Dot Product

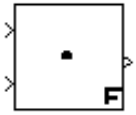
Purpose

Generate the dot product

Library

Math

Description



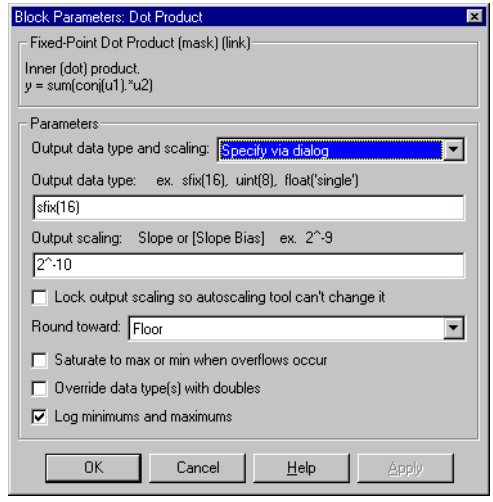
The Dot Product block is a masked S-function that generates the dot product of its two input vectors. The scalar output, `y`, is equal to the MATLAB operation

```
y = sum(conj(u1).* u2)
```

where `u1` and `u2` represent the inputs. If both inputs are vectors, they must be the same length.

For a detailed description of all block parameters, refer to “Block Parameters” on page 10-4. For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-27.

Parameters and Dialog Box



Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

Filter Direct Form I

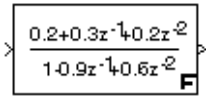
Purpose

Implement a Direct Form I realization of a filter

Library

Filters

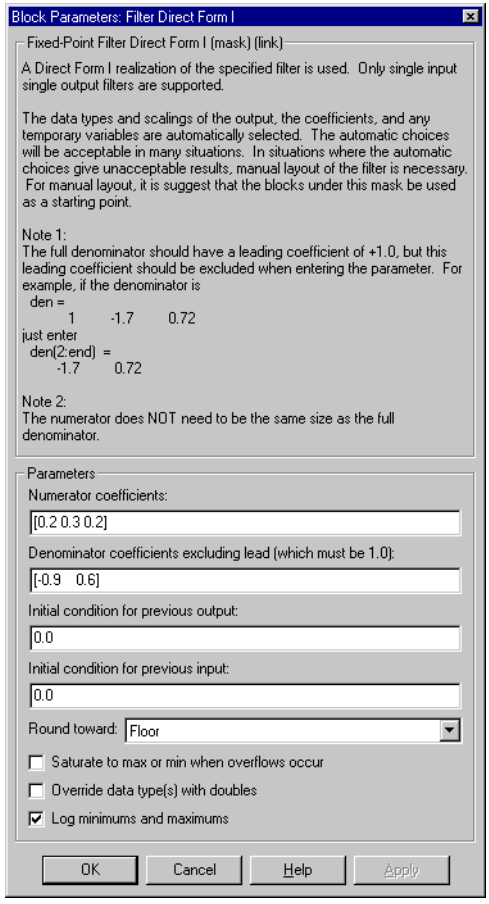
Description



The Filter Direct Form I block implements a Direct Form I realization of the filter specified by the **Numerator coefficients** and the **Denominator coefficients excluding lead** parameters. The block only supports single input-single output filters.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Parameters and Dialog Box



Numerator coefficients

Coefficients for the numerator of the filter.

Denominator coefficients excluding lead

Coefficients for the denominator of the filter, excluding the leading coefficient, which must be 1.0.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous input

Set the initial condition for the previous input.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

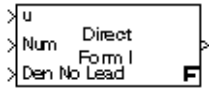
Purpose

Implement a time varying Direct Form I realization of a filter

Library

Filters

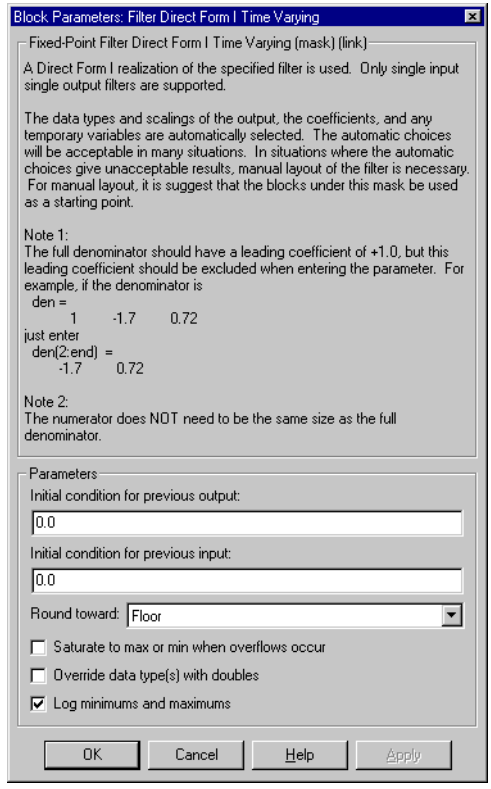
Description



The Filter Direct Form I Time Varying block implements a Direct Form I realization of the specified filter. The block only supports single input-single output filters.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Parameters and Dialog Box



Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous input

Set the initial condition for the previous input.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port u	Any data type supported by the blockset – it must be a scalar
Input Port Num	Any data type supported by the blockset – it must be a scalar
Input Port Den No Lead	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

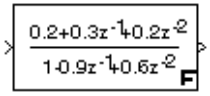
Purpose

Implement a Direct Form II realization of a filter

Library

Filters

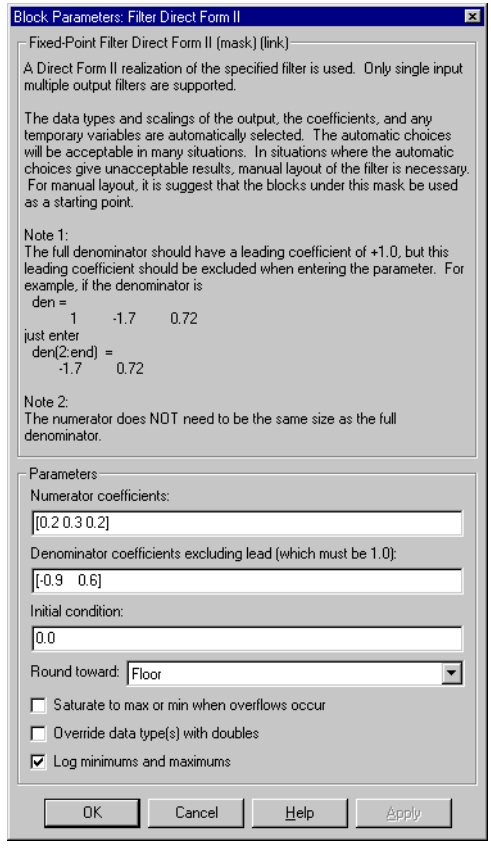
Description



The Filter Direct Form II block implements a Direct Form II realization of the filter specified by the **Numerator coefficients** and the **Denominator coefficients excluding lead** parameters. The block only supports single input-single output filters.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Parameters and Dialog Box



Numerator coefficients

Coefficients for the numerator of the filter.

Denominator coefficients excluding lead

Coefficients for the denominator of the filter, excluding the leading coefficient, which must be 1.0.

Initial condition

Set the initial condition.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

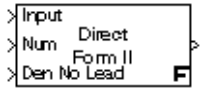
Purpose

Implement a time varying Direct Form II realization of a filter

Library

Filters

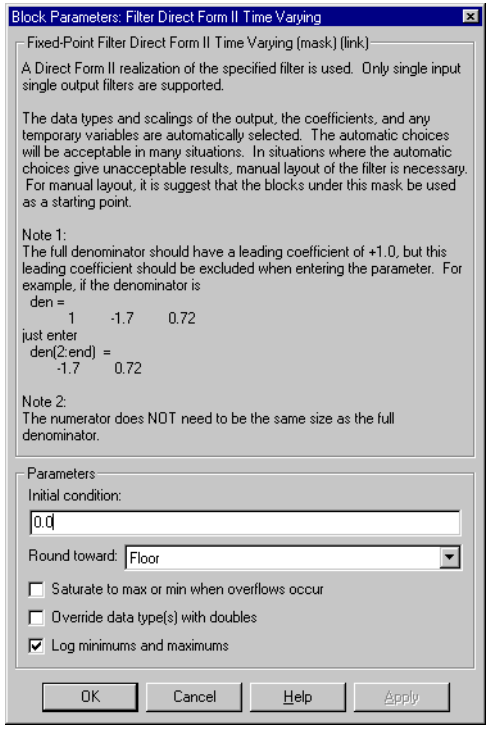
Description



The Filter Direct Form II Time Varying block implements a Direct Form II realization of the specified filter. The block only supports single input-single output filters.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Parameters and Dialog Box



Initial condition

Set the initial condition.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port u	Any data type supported by the blockset – it must be a scalar
Input Port Num	Any data type supported by the blockset – it must be a scalar
Input Port Den No Lead	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

Purpose

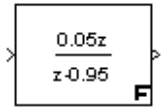
Implement a discrete-time first order filter

Library

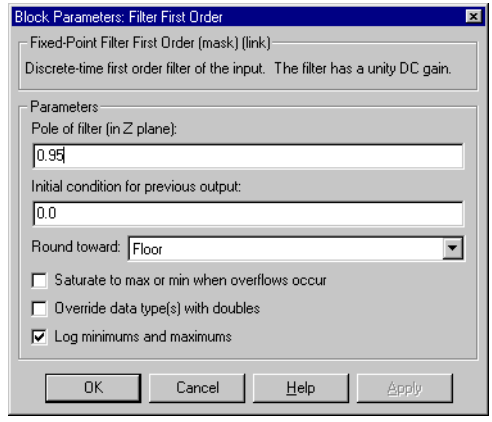
Filters

Description

The Filter First Order block implements a discrete-time first order filter of the input. The filter has a unity DC gain.



Parameters and Dialog Box



Pole of filter (in Z plane)

Set the pole of the filter.

Initial condition for previous output

Set the initial condition for the previous output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

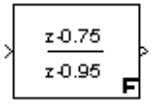
Purpose

Implement a discrete-time lead or lag filter

Library

Filters

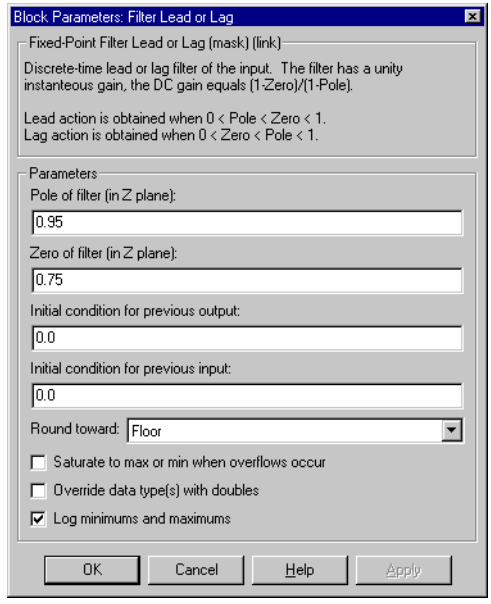
Description



The Filter Lead or Lag block implements a discrete-time lead or lag filter of the input. The instantaneous gain of the filter is one, and the DC gain is equal to (1-z)/(1-p), where z is the zero and p is the pole of the filter.

The block implements a lead filter when $0 < z < p < 1$, and implements a lag filter when $0 < p < z < 1$.

Parameters and Dialog Box



Pole of filter (in Z plane)

Set the pole of the filter.

Zero of filter (in Z plane)

Set the zero of the filter.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous input

Set the initial condition for the previous input.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

See Also

FIR, and related blocks

Purpose

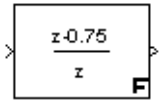
Implement a discrete-time filter that has a real zero and no pole

Library

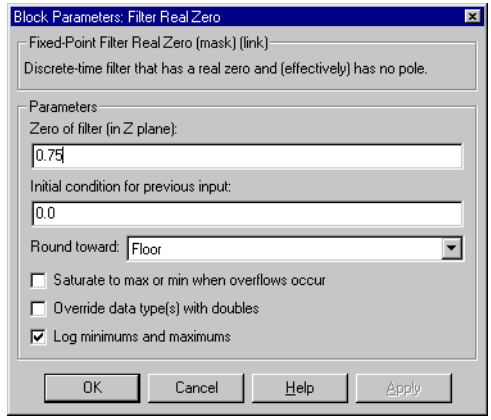
Filters

Description

The Filter Real Zero block implements a discrete-time filter that has a real zero and effectively has no pole.



Parameters and Dialog Box



Zero of filter (in Z plane)

Set the zero of the filter.

Initial condition for previous input

Set the initial condition for the previous input.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

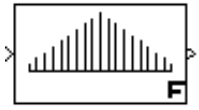
See Also

FIR, and related blocks

Purpose

Implement a fixed-point finite impulse response (FIR) filter

Description



The FIR block is a masked S-function that samples and holds the N most recent inputs, multiplies each input by a specified value (its FIR coefficient), and stacks them in a vector. This block supports both single-input/single-output (SISO) and single-input/multi-output (SIMO) modes.

For the SISO mode, the **FIR coefficients** parameter is specified as a row vector. For the SIMO mode, the **FIR coefficients** are specified as a matrix where each row corresponds to a separate output.

The **Initial condition** parameter provides the initial values for all times preceding the start time in the FIR realization. You specify the time interval between samples with the **Sample time** parameter.

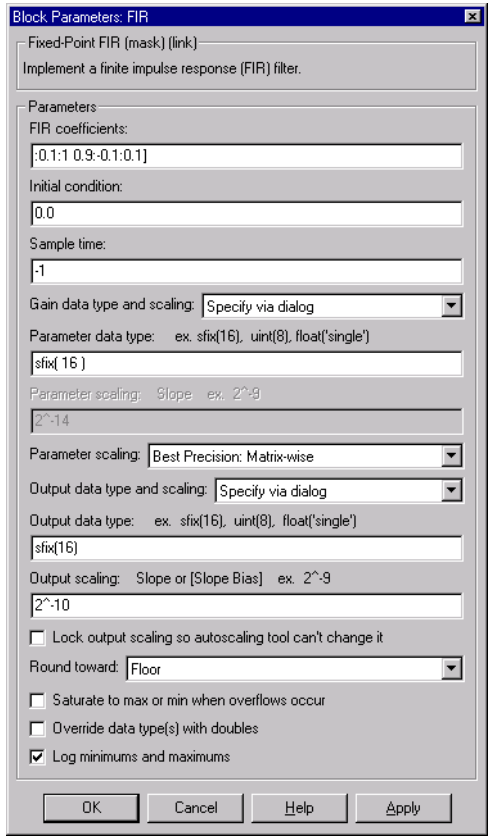
You specify the scaling for the FIR coefficients with the **Parameter scaling** parameter. Note that there are two dialog box parameters that control the FIR coefficient scaling: one associated with an edit field, and one associated with a parameter list. If **Parameter data type** is a generalized fixed-point number such as `sfix(16)`, the **Parameter scaling** list provides you with these scaling modes:

- `Use Specified Scaling` – This mode uses the slope/bias or radix point-only scaling specified for the editable **Parameter scaling** parameter (for example, `2^-10`).
- `Best Precision: Element-wise` – This mode produces radix points such that the precision is maximized for each element of the **FIR coefficients** parameter.
- `Best Precision: Row-wise` – This mode produces a common radix point for each element of the **FIR coefficients** row based on the best precision for the largest value of that row.
- `Best Precision: Column-wise` – This mode produces a common radix point for each element of the **FIR coefficients** column based on the best precision for the largest value of that column.
- `Best Precision: Matrix-wise` – This mode produces a common radix point for each element of the **FIR coefficients** matrix based on the best precision for the largest value of the matrix.

If the FIR coefficients are specified as a row vector, then scaling element-wise and column-wise produce the same result, while scaling matrix-wise and row-wise produce the same result.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Parameters and Dialog Box



FIR coefficients

FIR coefficients. One row per output.

Initial condition

Initial values for all times preceding the start time.

Sample time

Sample time.

Parameter data type

Any data type supported by the Fixed-Point Blockset.

Parameter scaling

Radix point-only or slope/bias scaling. Additionally, the **FIR coefficients** vector or matrix can be scaled using the constant vector or constant matrix scaling modes for maximizing precision. These scaling modes are available only for generalized fixed-point data types.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Conversions and Operations

The **FIR coefficients** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation.The **Initial condition** parameter is converted from doubles to the input data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

The FIR block first multiplies its inputs by the **FIR coefficients** parameter, converts those results to the output data type using the specified rounding and overflow modes, and then carries out the summation. Refer to "Rules for Arithmetic Operations" on page 4-30 for more information about the rules this block adheres to when performing operations.

Example

Suppose you want to configure this block for two outputs (SIMO mode) where the first output is given by

$$y_1(k) = a_1 \cdot u(k) + b_1 \cdot u(k-1) + c_1 \cdot u(k-2)$$

the second output is given by

$$y_2(k) = a_2 \cdot u(k) + b_2 \cdot u(k-1)$$

and the initial values of $u(k-1)$ and $u(k-2)$ are given by `ic1` and `ic2`, respectively. To configure the FIR block for this situation, you must specify the **FIR coefficient** parameter as `[a1 b1 c1; a2 b2 c2]` where `c2` = 0, and the **Initial condition** parameter as `[ic1 ic2]`.

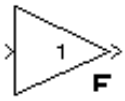
Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

Purpose

Multiply the input by a constant

Description



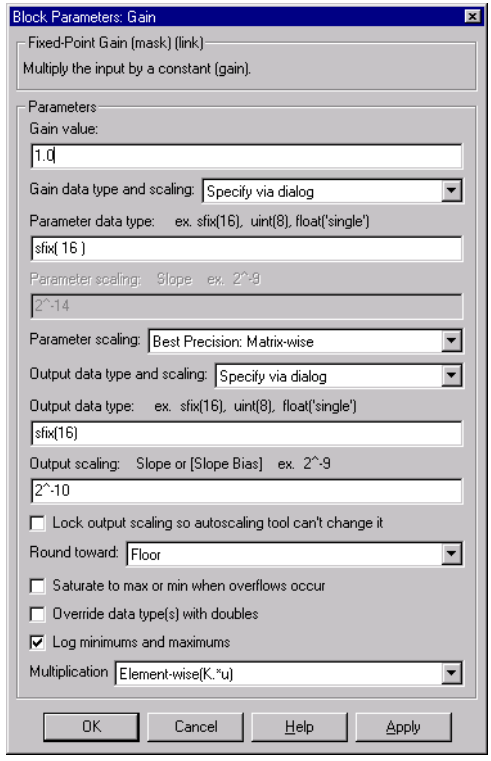
The Gain block is a masked S-function that multiplies the input by a constant value (referred to as the gain). To multiply the input by a constant matrix, use the Matrix Gain block.

You specify the gain with the **Gain value** parameter. The gain can be a scalar or a vector. You specify the scaling for the gain with the **Parameter scaling** parameter. Note that there are two dialog box parameters that control the gain scaling: one associated with an edit field, and one associated with a parameter list. If **Parameter data type** is a generalized fixed-point number such as `sfix(16)`, the **Parameter scaling** list provides you with these scaling modes:

- `Use Specified Scaling` – This mode uses the slope/bias or radix point-only scaling specified for the editable **Parameter scaling** parameter (for example, `2^-10`).
- `Best Precision: Element-wise` – This mode produces radix points such that the precision is maximized for each element of the **Gain value** vector.
- `Best Precision: Vector-wise` – This mode produces a common radix point for each element of the **Gain value** vector based on the best precision for the largest value of the vector.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

Parameters and Dialog Box



Gain value

Specify as a vector or scalar.

Parameter data type

Any data type supported by the Fixed-Point Blockset.

Parameter scaling

Radix point-only or slope/bias scaling. Additionally, if **Gain value** is specified as a vector, it can be scaled using the constant vector scaling modes for maximizing precision. These scaling modes are available only for generalized fixed-point data types.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Parameter data type** and **Output data type** values are overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Conversions and Operations

The **Gain value** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

The Gain block first multiples its inputs by the **Gain value** parameter, and then converts those results to the output data type using the specified rounding and overflow modes. Refer to "Rules for Arithmetic Operations" on page 4-30 for more information about the rules this block adheres to when performing operations.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

Gateway In

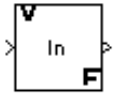
Purpose

Convert a Simulink data type to a Fixed-Point Blockset data type

Library

Data Type

Description

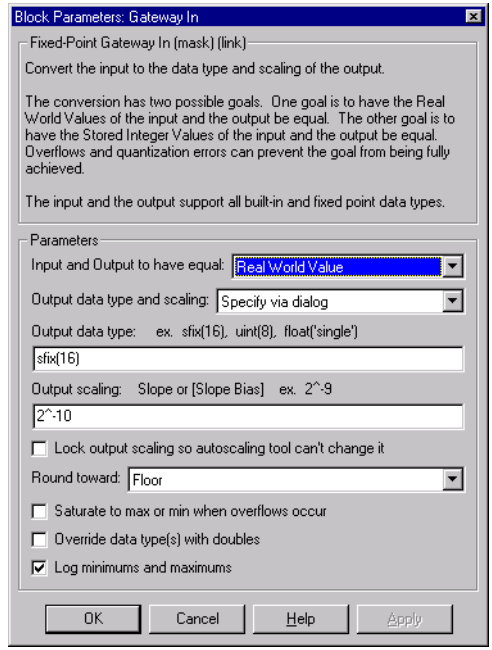


The Gateway In block is a masked S-function that converts a built-in Simulink data type to a Fixed-Point Blockset data type.

The **Input and Output to have equal** parameter list controls how the input is processed. The possible values are **Real World Value** and **Stored Integer**. In terms of the general encoding scheme described in "Scaling" on page 3-5, **Real World Value** treats the input as $V = SQ + B$ where S is the slope and B is the bias. V is used to produce $Q = (V - B)/S$, which is stored in the output. **Stored Integer** treats the input as a stored integer, Q. The value of Q is directly used to produce the output. In this mode, the input and output are identical except that the input is a raw integer lacking proper scaling information. In both modes, the output data type includes the scaling information needed to correctly interpret the signal as a real-world value.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Parameters and Dialog Box



Input and Output to have equal

Specify the type of value that the input and output are to have equal.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

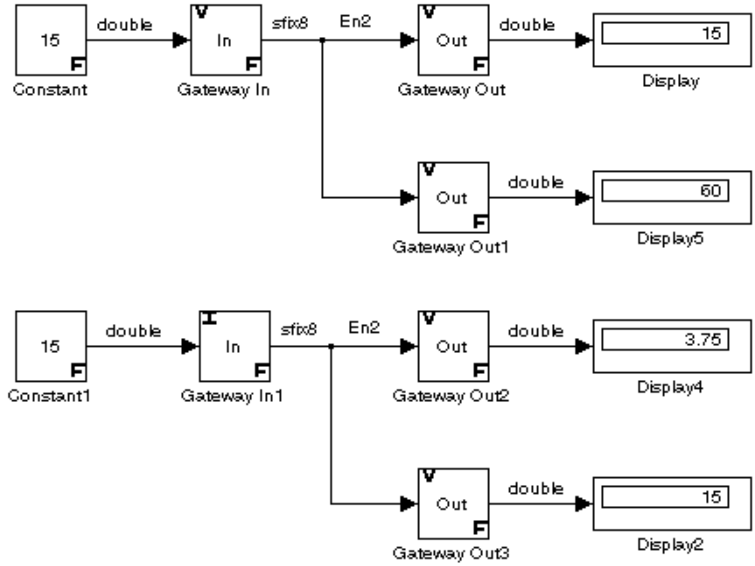
If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Example

This example uses the Gateway In block to help you understand the difference between a real-world value and a stored integer. Consider the two fixed-point models shown below.

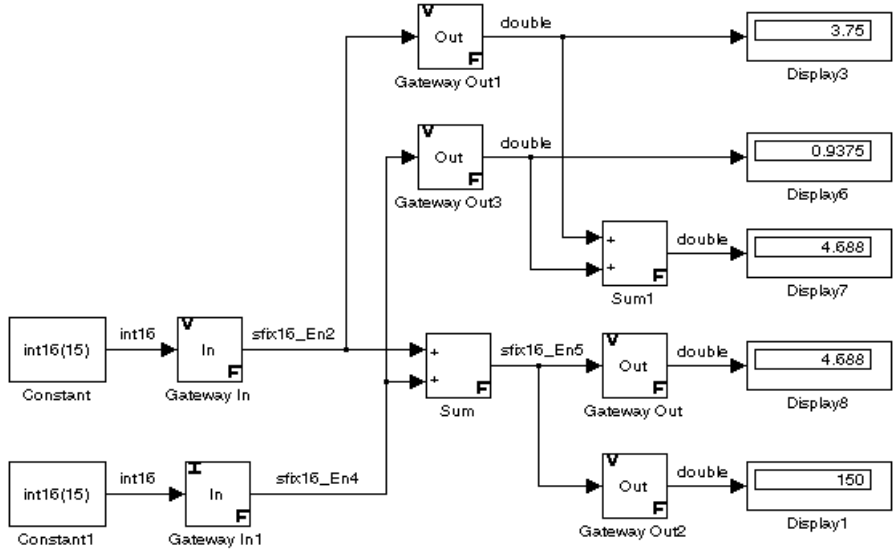


In the top model, the Gateway In block treats the input as a real-world value, and maps that value to an 8-bit signed generalized fixed-point data type with a scaling of 2^{-2}. If the value is output from the Gateway Out block as a

real-world value, then the scaling and data type information is retained and the output value is 001111.00, or 15. If the value is output from the Gateway Out block as a stored integer, then the scaling and data type information is not retained and the stored integer is interpreted as 00111100, or 60.

In the bottom model, the Gateway In block treats the input as a stored integer, and the data type and scaling information is not applied. If the value is output from the Gateway Out block as a real-world value, then the scaling and data type information is applied to the stored integer, and the output value is 000011.11, or 3.75. If the value is output from the Gateway Out block as a stored integer, then you get back the original input value of 15.

The model shown below illustrates how a summation operation applies to real-world values and stored integers, and how scaling information is dealt with in generated code.



Note that the summation operation produces the correct result when the Gateway Out block outputs a real-world value. This is because the specified scaling information is applied to the stored integer value. However, when the Gateway Out block outputs a stored integer value, then the summation

operation produces an unexpected result due to the absence of scaling information.

If you generate code for the above model, then the code captures the appropriate scaling information. The code for the Sum block is shown below. The inputs to this block are tagged with the specified scaling information so that the necessary shifts are performed for the summation operation.

```
/* Sum Block: <Root>/Sum
   *
   *  y =  u0 + u1
   *
   * Input0  Data Type:  Fixed Point    S16  2^-2
   * Input1  Data Type:  Fixed Point    S16  2^-4
   * Output0 Data Type:  Fixed Point    S16  2^-5
   *
   * Round Mode: Floor
   * Saturation Mode: Wrap
   *
   */
  sum = ((in1) << 3);
  sum += ((in2) << 1);
```

Characteristics

Input Port	Any built-in Simulink data type
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	No

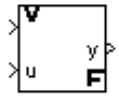
Purpose

Convert a Simulink data type to a Fixed-Point Blockset data type, and inherit the data type and scaling

Library

Data Type

Description



The Gateway In Inherited block is a masked S-function that converts a built-in Simulink data type to a Fixed-Point Blockset data type.

The block requires two inputs. The first (top) input provides the data type and scaling information. The second (bottom) input passes through to the output, and inherits the data type and scaling of the first input. If you want to explicitly specify the output data type and scaling, use the Gateway In block.

The **Input and Output to have equal** parameter list controls how the input is processed. The possible values are **Real World Value** and **Stored Integer**. In terms of the general encoding scheme described in "Scaling" on page 3-5, **Real World Value** treats the input as $V = SQ + B$ where S is the slope and B is the bias. **Stored Integer** treats the input as a stored integer, Q. For more information about this parameter list, refer to the Gateway In block.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

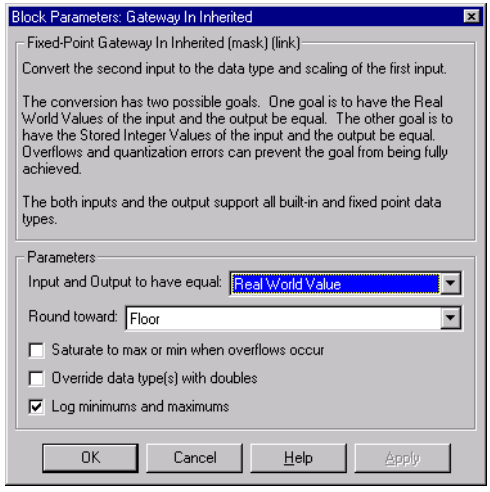
Remarks

Inheriting the data type and scaling provides these advantages:

- It makes reusing existing models easier.
- It allows you to create new fixed-point models with less effort since you can avoid the detail of specifying the associated parameters.

Gateway In Inherited

Parameters and Dialog Box



Input and Output to have equal

Specify the type of value that the input and output are to have equal.

Round toward

Rounding mode for fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the output data type is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port	Any built-in Simulink data type
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Gateway In, and related blocks

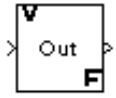
Purpose

Convert a Fixed-Point Blockset data type to a Simulink data type

Library

Data Type

Description



The Gateway Out block is a masked S-function that converts any data type supported by the Fixed-Point Blockset to a Simulink data type.

The **Output and Input to have equal** parameter list controls how the output is treated. The possible values are **Real World Value** and **Stored Integer**. In terms of the general encoding scheme described in "Scaling" on page 3-5, **Real World Value** treats the output as $V = SQ + B$ where S is the slope and B is the bias. **Stored Integer** treats the output as a stored integer, Q. Selecting **Stored Integer** may be useful in these circumstances:

- If you are generating code for a fixed-point processor, the resulting code only uses integers and does not use floating-point operations.
- If you want to partition your model based on hardware characteristics. For example, part of your model may involve simulating hardware that produces integers as output.

Note If the fixed-point signal is a true integer such as `sint(8)` or `uint(16)`, then **Real World Value** and **Stored Integer** produce identical output values.

For more information about this parameter list, refer to the Gateway In block description.

The **Output data type** parameter list specifies the Simulink data type to use for the output. All built-in data types are supported as well as the `boolean` data type. `auto` indicates the Fixed-Point Blockset data type is converted to whatever data type Simulink back propagates.

Remarks

MATLAB's built-in integer data types are limited to 32 bits. If you want to output fixed-point numbers that range between 33 and 53 bits without loss of precision or range, you should use the Gateway Out block to store the value inside a double.

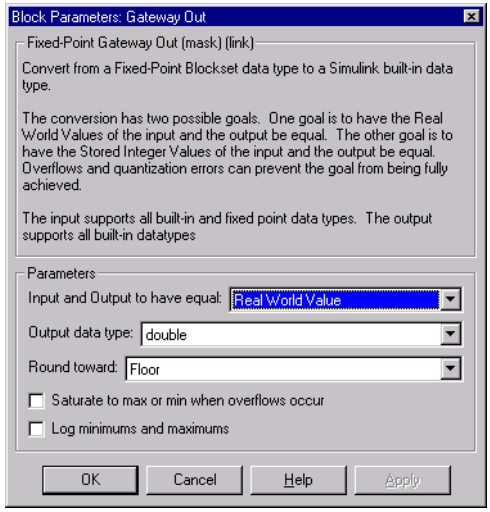
If you want to output fixed-point numbers with more than 53 bits without loss of precision or range, then you must break the number into pieces using the Gain block, and then output the pieces using the Gateway Out block.

For example, suppose the original signal is an unsigned 128-bit value with default scaling. You can break this signal into four pieces using four parallel Gain blocks configured with the gain and output settings shown below.

Piece	Gain	Output Data Type
1	`2^0`	`uint(32)` – Least significant 32 bits
2	`2^-32`	`uint(32)`
3	`2^-64`	`uint(32)`
4	`2^-96`	`uint(32)` – Most significant 32 bits

For each Gain block, you must also configure the **Round toward** parameter to `Floor`, and the **Saturate to max or min when overflows occur** check box must be unchecked.

Parameters and Dialog Box



Output and Input to have equal

Specify the type of value the input and output are to have equal.

Output data type

Any built-in data type supported by Simulink.

Round toward

Rounding mode for fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any built-in Simulink data type
Direct Feedthrough	Yes
Scalar Expansion	N/A

See Also

Gateway In, and related blocks

Increment Real World

Purpose

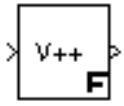
Increase the real world value of the signal by one

Library

Math

Description

The Increment Real World block is a masked block that increases the real world value of the signal by one. Overflows always wrap.



Parameters and Dialog Box

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

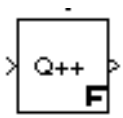
Purpose

Increase the stored integer value of a signal by one

Library

Math

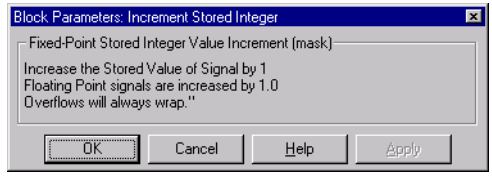
Description



The Increment Stored Integer block is a masked block that increases the stored integer value of a signal by one.

Floating-point signals are also increased by one, and overflows always wrap.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Increment Real World

Integer Delay

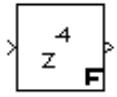
Purpose

Delay a signal N sample periods

Library

Delays & Holds

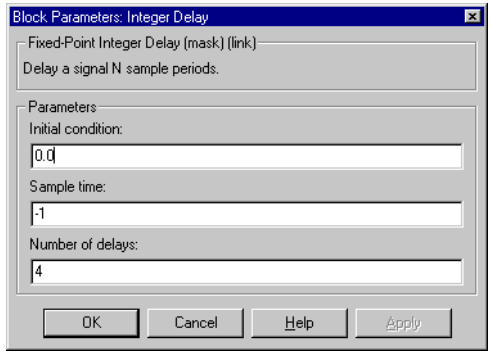
Description



The Integer Delay block delays its input by N sample periods.

The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are delayed by the same sample period.

Parameters and Dialog Box



Initial condition

The initial output of the simulation.

Sample time

Sample time.

Number of delays

The number of periods to delay the input signal.

Conversions

The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	No
Scalar Expansion	Of input or initial conditions

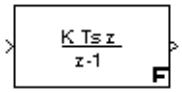
Purpose

Perform discrete-time integration of a signal using the backward method

Library

Calculus

Description

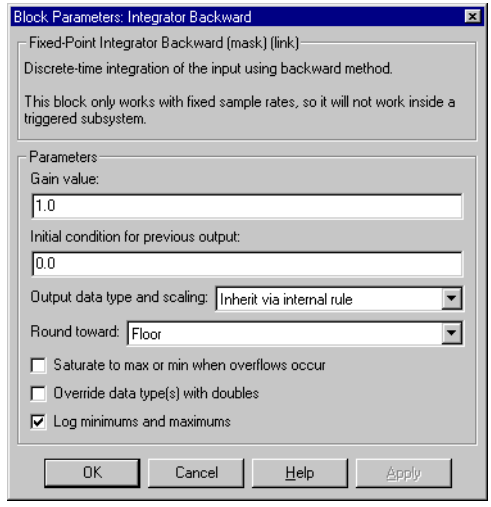


The Integrator Backward block performs a discrete-time integration of a signal using the backward method. The block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero. The block outputs the sum up to the nth time step at time n.

Remarks

The output of the Integrator Backward block differs from the output of the Integrator Forward block only by the first and last terms in the cumulative sum.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

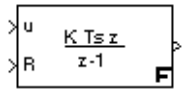
Purpose

Perform discrete-time integration of a signal using the backward method with external Boolean reset

Library

Calculus

Description



The Integrator Backward Resettable block performs a discrete-time integration of a signal using the backward method.

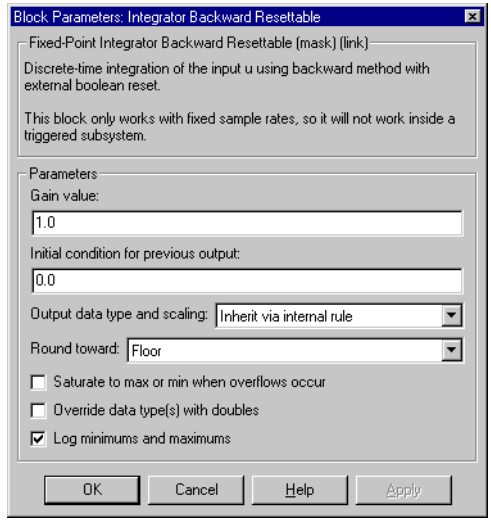
The block can reset its state based on an external reset signal R. When the reset signal R is false, the block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero.

When the reset signal R is true, the block outputs the **Initial condition for previous output** parameter.

Remarks

The output of the Integrator Backward Resettable block differs from the output of the Integrator Forward Resettable block only by the first and last terms in the cumulative sum.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

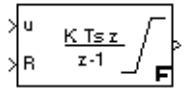
Purpose

Perform discrete-time limited integration of a signal using the backward method, with external Boolean reset

Library

Calculus

Description



The Integrator Backward Resettable Limited block performs a discrete-time integration of a signal using the backward method.

The block can reset its state based on an external reset signal R. When the cumulative sum reaches one of the limits given by the **Upper limit** and **Lower limit** parameters, the sum saturates to that limit.

When the reset signal R is false, the block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero.

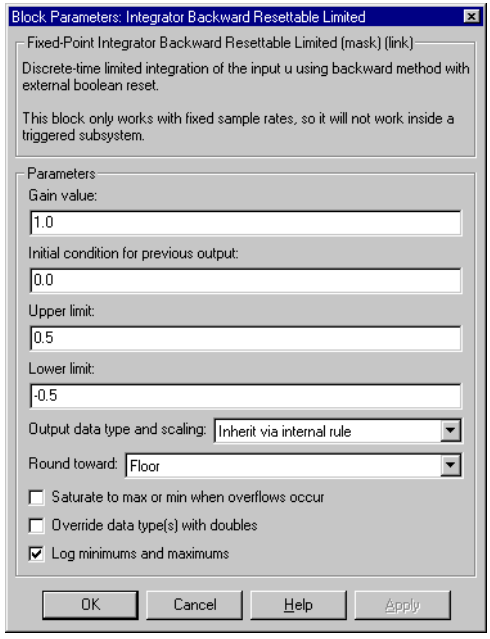
When the reset signal R is true, the block outputs the **Initial condition for previous output** parameter.

Remarks

The output of the Integrator Backward Resettable Limited block differs from the output of the Integrator Forward Resettable Limited block only by the first and last terms in the cumulative sum.

Integrator Backward Resettable Limited

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Upper limit

The upper limit for saturation of the cumulative sum.

Lower limit

The lower limit for saturation of the cumulative sum.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

Integrator Forward

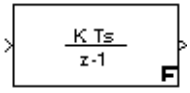
Purpose

Perform discrete-time integration of a signal using the forward method

Library

Calculus

Description

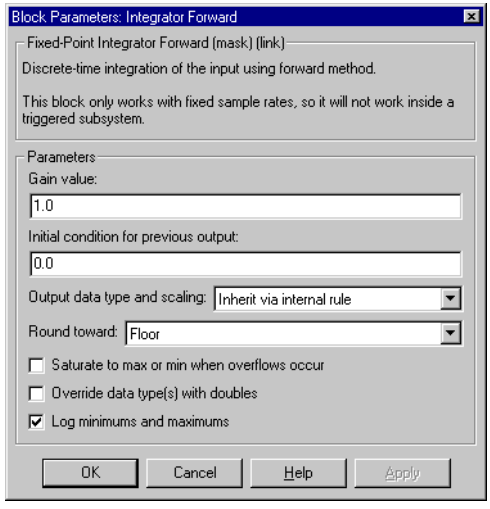


The Integrator Forward block performs a discrete-time integration of a signal using the forward method. The block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero. The block outputs the sum up to the nth time step at time n+1. The first term of the sum is the **Initial condition for previous output** parameter.

Remarks

The output of the Integrator Forward block differs from the output of the Integrator Backward block only by the first and last terms in the cumulative sum.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

Integrator Forward Resettable

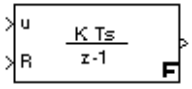
Purpose

Perform discrete-time integration of a signal using the forward method, with external Boolean reset

Library

Calculus

Description



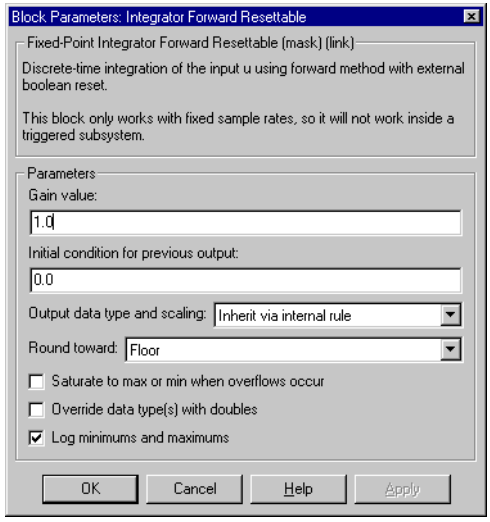
The Integrator Forward Resettable block performs a discrete-time integration of a signal using the forward method. When the external reset signal R is false, the block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero.

When the external reset signal R is true, the block outputs the **Initial condition for previous output** parameter.

Remarks

The output of the Integrator Forward Resettable block differs from the output of the Integrator Backward Resettable block only by the first and last terms in the cumulative sum.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

Integrator Forward Resettable Limited

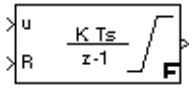
Purpose

Perform discrete-time limited integration of a signal using the forward method, with external Boolean reset

Library

Calculus

Description



The Integrator Forward Resettable Limited block performs a discrete-time integration of a signal using the forward method. When the cumulative sum reaches one of the limits given by the **Upper limit** and **Lower limit** parameters, the sum saturates to that limit.

When the external reset signal R is false, the block multiplies the input by the weighted sample time and adds the result to the cumulative sum since time zero.

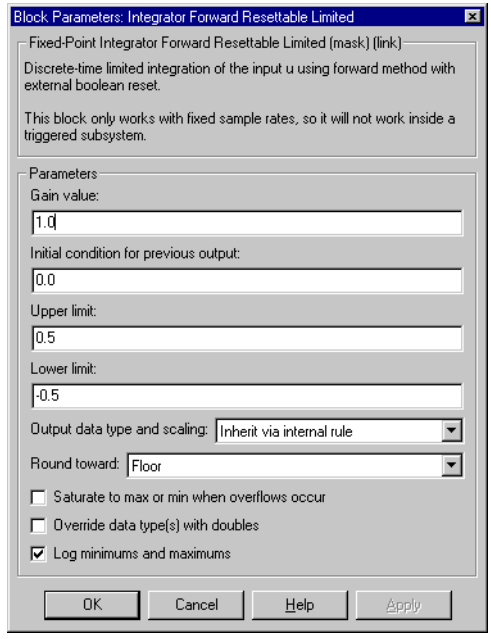
When the external reset signal R is true, the block outputs the **Initial condition for previous output** parameter.

The first term of the sum is the product of the weighted sample time and the value of the **Initial condition for previous input parameter**.

Remarks

The output of the Integrator Forward Resettable Limited block differs from the output of the Integrator Backward Resettable Limited block only by the first and last terms in the cumulative sum.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Upper limit

The upper limit for saturation of the cumulative sum.

Lower limit

The lower limit for saturation of the cumulative sum.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

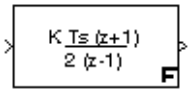
Purpose

Perform discrete-time integration of a signal using the trapezoidal method

Library

Calculus

Description



The Integrator Trapezoidal block performs a discrete-time integration of a signal using the trapezoidal method. At time step k, the block computes the average of the inputs at times k-1 and k, multiplies the average by the weighted sample time, and adds the result to the cumulative sum since time zero. The block outputs the sum up to the k'th time step at time.

The block calculates the output at time k by the rule

$$y(k) = y(k-1) + w(k) + w(k-1)$$

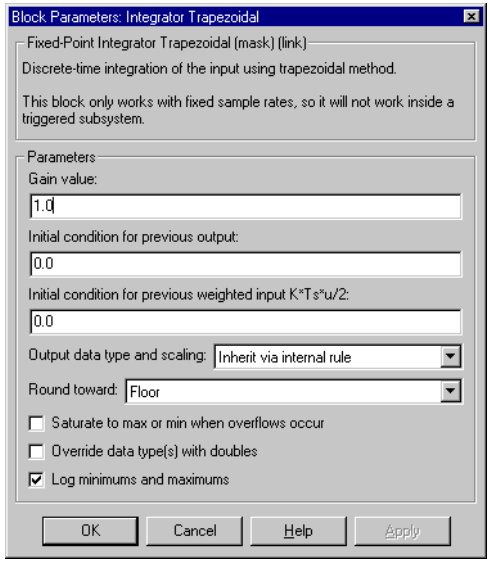
where $u(k)$ is the input at time k and

$$w(k) = \frac{K \cdot Ts}{2} \cdot u(k)$$

At the first time step, $y(0)$ is set to the value of **Initial condition for previous output**, and $w(0)$ is set to the value of **Initial condition for previous weighted input K*Ts*u/2**.

Integrator Trapezoidal

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous weighted input K*Ts*u/2

Set the initial condition for the previous weighted input.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

Integrator Trapezoidal Resettable

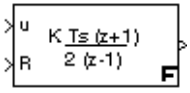
Purpose

Perform discrete-time integration of a signal using the trapezoidal method, with external Boolean reset

Library

Calculus

Description



The Integrator Trapezoidal Resettable block performs a discrete-time integration of a signal using the trapezoidal method.

The block can reset its state based on an external reset signal R. When the reset signal R is false at time k, the block calculates the output at time k by the rule

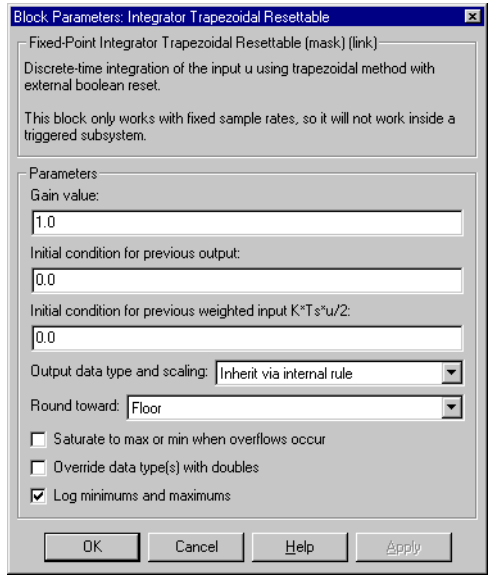
$$y(k) = y(k-1) + w(k) + w(k-1)$$

where $u(k)$ is the input at time k and

$$w(k) = \frac{K \cdot Ts}{2} \cdot u(k)$$

When the reset signal R is true at time k, the block resets the output *y(k)* to the value of the **Initial condition for previous output** parameter, and resets *w(k)* to the value of the **Initial condition for previous weighted input K*Ts*u/2** parameter.

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous weighted input K*Ts*u/2

Set the initial condition for the previous weighted input.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

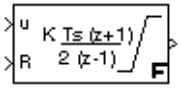
Purpose

Perform discrete-time limited integration of a signal using the trapezoidal method, with external Boolean reset

Library

Calculus

Description



The Integrator Trapezoidal Resettable block performs a discrete-time integration of a signal using the trapezoidal method.

The block can reset its state based on an external reset signal R. When the cumulative sum reaches one of the limits given by the **Upper limit** and **Lower limit** parameters, the sum saturates to that limit.

When the reset signal R is false at time step k, the block calculates the output at time k by the rule

$$y(k) = y(k-1) + w(k) + w(k-1)$$

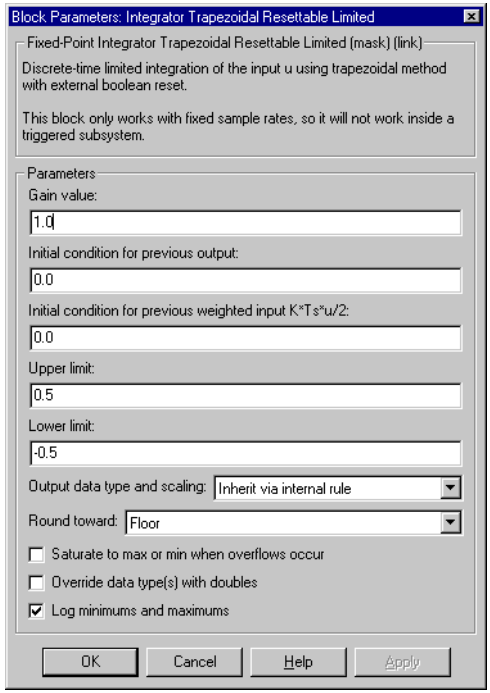
where $u(k)$ is the input at time k and

$$w(k) = \frac{K \cdot Ts}{2} \cdot u(k)$$

When the reset signal R is true at time k, the block resets the output *y(k)* to the value of the **Initial condition for previous output** parameter. The block also resets *w(k)* to the value of the **Initial condition for previous weighted input K*Ts*u/2** parameter.

Integrator Trapezoidal Resettable Limited

Parameters and Dialog Box



Gain value

Specify the weight by which the sample time is multiplied.

Initial condition for previous output

Set the initial condition for the previous output.

Initial condition for previous weighted input K*Ts*u/2

Set the initial condition for the previous weighted input.

Upper limit

The upper limit for saturation of the cumulative sum.

Lower limit

The lower limit for saturation of the cumulative sum.

Output data type and scaling

The options are:

- **Specify via dialog**
- **Inherit via internal rule**
- **Inherit via back propagation**

When **Specify via dialog** is selected, you can specify the **Output data type** and **Output scaling** parameters.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	Yes, of the input and reset source ports
Scalar Expansion	Of inputs and gain

See Also

Integrator Backward, and related blocks

Interval Test

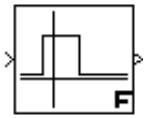
Purpose

Determine if a signal is in a specified interval

Library

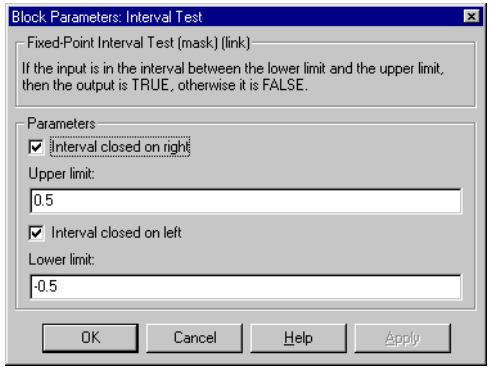
Logic & Comparison

Description



The Interval Test block outputs TRUE if the input is between the values specified by the **Lower limit** and **Upper limit** parameters. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the **Lower limit** or the **Upper limit** is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are checked in the dialog box.

Parameters and Dialog Box



Interval closed on right

When the box is checked, the **Upper limit** is included in the interval for which the block outputs TRUE.

Upper limit

The upper limit of the interval for which the block outputs TRUE.

Interval closed on left

When the box is checked, the **Lower limit** is included in the interval for which the block outputs TRUE.

Lower limit

The lower limit of the interval for which the block outputs TRUE.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as input
Direct Feedthrough	Yes
Scalar Expansion	Yes

Interval Test Dynamic

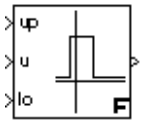
Purpose

Determine if a signal is in a specified interval

Library

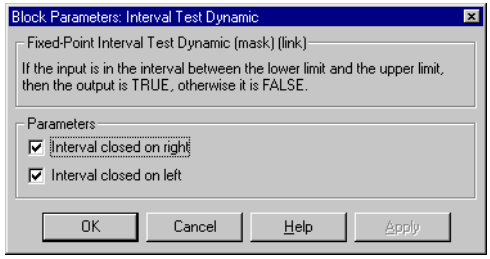
Logic & Comparison

Description



The Interval Test Dynamic block outputs TRUE if the input is between the values of the external signals up and lo. The block outputs FALSE if the input is outside those values. The output of the block when the input is equal to the signal up or the signal lo is determined by whether the boxes next to **Interval closed on left** and **Interval closed on right** are checked in the dialog box.

Parameters and Dialog Box



Interval closed on right

When the box is checked, the **Upper limit** is included in the interval for which the block outputs TRUE.

Interval closed on left

When the box is checked, the **Lower limit** is included in the interval for which the block outputs TRUE.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as output
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Interval Test

Purpose

Perform the specified logical operation on the inputs

Library

Logic & Comparison

Description

The Logical Operation block is a masked S-function that performs the specified logical operation on its inputs. An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

You select the Boolean operation connecting the inputs with the **Operator** parameter list. The supported operations are given below.

Operation	Description
AND	TRUE if all inputs are TRUE
OR	TRUE if at least one input is TRUE
NAND	TRUE if at least one input is FALSE
NOR	TRUE when no inputs are TRUE
XOR	TRUE if an odd number of inputs are TRUE
NOT	TRUE if the input is FALSE

The number of input ports is specified with the **Number of input ports** parameter. The output type is specified with the **Logical output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

Note The output data type should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers, and any floating-point data type.

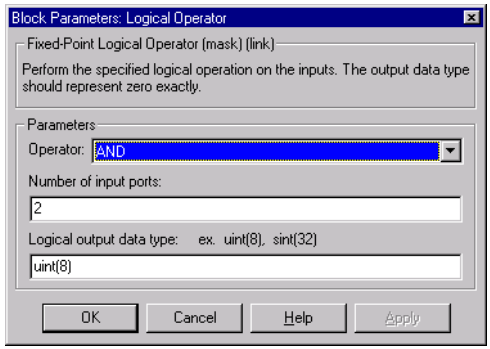
The size of the output depends on the number of inputs, their vector size, and the selected operator:

- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the input vector elements.

- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. The output is always a scalar.
- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the operation is performed between corresponding elements of the vectors to produce a vector output.

When configured as a multi-input XOR gate, this block performs an addition-modulo-two operation as mandated by the IEEE Standard for Logic Elements.

Parameters and Dialog Box



Operator

Logical operator used to connect the inputs.

Number of input ports

Number of inputs.

Logical output data type

Output data type. You should only use data types that represent zero exactly.

Characteristics

Input Port(s)	Any data type supported by the blockset
Output Port	Any data type supported by the blockset that can exactly represent zero
Direct Feedthrough	Yes
Scalar Expansion	Of inputs

Purpose

Approximate a one-dimensional function using a selected look-up method

Library

Lookup

Description

The Look-Up Table block is a masked S-function that computes an approximation to some function `y=f(x)` given `x`, `y` data vectors. The look-up method can use interpolation, extrapolation, or the original values of the input.

The length of the `x` and `y` data vectors provided to this block must match. Also, the `x` data vector must be strictly monotonically increasing after conversion to the input's fixed-point data type. Note that due to quantization, the `x` data vector may be strictly monotonic in doubles format, but not so after conversion to a fixed-point data type. To map two inputs to an output, use the Look-Up Table (2D) block.

You define the table by specifying the **Vector of input values** parameter as a 1-by-n vector and the **Vector of output values** parameter as a 1-by-n vector. The block generates output based on the input values using one of these methods selected from the **Method** parameter list:

- `Interpolation-Extrapolation` – This is the default method; it performs linear interpolation and extrapolation of the inputs.
  - If a value matches the block's input, the output is the corresponding element in the output vector.
  - If no value matches the block's input, then the block performs linear interpolation between the two appropriate elements of the table to determine an output value. If the block input is less than the first or greater than the last input vector element, then the block extrapolates using the first two or last two points.
- `Interpolation-Use End Values` – This method performs linear interpolation as described above but does not extrapolate outside the end points of the input vector. Instead, the end-point values are used.
- `Use Input Nearest` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest the current input is found. The corresponding element in `y` is then used as the output.
- `Use Input Below` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest and below the current input is found. The

corresponding element in `y` is then used as the output. If there is no element in `x` below the current input, then the nearest element is found.

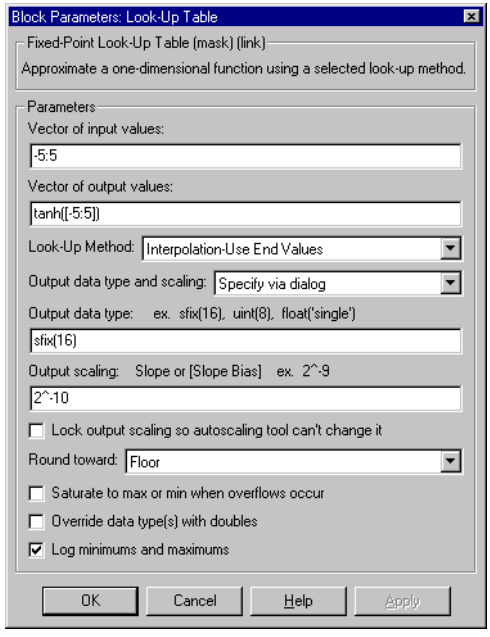
- `Use Input Above` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest and above the current input is found. The corresponding element in `y` is then used as the output. If there is no element in `x` above the current input, then the nearest element is found.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Remarks

To avoid parameter saturation errors, the automatic scaling script `autofixexp` employs a special rule for the Look-Up Table block. `autofixexp` modifies the scaling by using the output look-up values in addition to the logged minimum and maximum simulation values. This prevents the data from being saturated to different values. The look-up values are given by the **Vector of output values** parameter (the `YDataPoints` variable).

Parameters and Dialog Box



Vector of input values

The vector of input values must be the same size as the output vector and strictly monotonically increasing.

Vector of output values

The vector of output values must be the same size as the input vector.

Look-up method

Look-up method.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

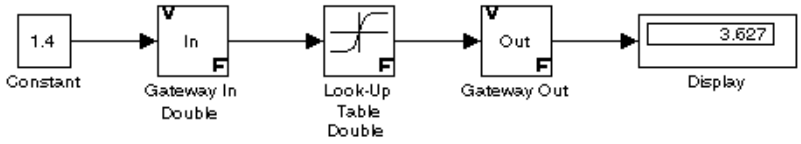
Conversions

The **Vector of input values** parameter is converted from doubles to the input data type. The **Vector of output values** parameter is converted from doubles to the output data type. Both conversion are performed offline using

round-to-nearest and saturation. Refer to “Parameter Conversions” on page 4-27 for more information about parameter conversions.

Example

Suppose the Look-Up Table block is configured to use a vector of input values given by `[-5:5]`, and a vector of output values given by `sinh([-5:5])`. Using the model shown below,



the following results were generated.

Look-Up Method	Input	Output	Comment
`Interpolation-Extrapolation`	`1.4`	`2.153`	N/A
	`5.2`	`83.59`	N/A
`Interpolation-Use End Values`	`1.4`	`2.153`	N/A
	`5.2`	`74.2`	The value for `sinh(5.0)` was used.
`Use Input Nearest`	`1.4`	`1.175`	The value for `sinh(1.0)` was used.
`Use Input Below`	`1.4`	`1.175`	The value for `sinh(1.0)` was used.
	`-5.2`	`-74.2`	The value for `sinh(-5.0)` was used.
`Use Input Above`	`1.4`	`3.627`	The value for `sinh(2.0)` was used.
	`5.2`	`74.2`	The value for `sinh(5.0)` was used.

Characteristics

Input Port(s)	Any data type supported by the blockset
Output Port	Any data type supported by the blockset

Direct Feedthrough	Yes
Scalar Expansion	No

Look-Up Table Dynamic

Purpose

Approximate a one-dimensional function using a selected look-up method and a dynamically specified table

Library

LookUp

Description

The Look-Up Table Dynamic block is a masked S-function that computes an approximation to some function `y=f(x)` given `x`, `y` data vectors. The look-up method can use interpolation, extrapolation, or the original values of the input.

The `x` data vector must be strictly monotonically increasing after conversion to the input's fixed-point data type. Note that due to quantization, the `x` data vector may be strictly monotonic in doubles format, but not so after conversion to a fixed-point data type.

Note Unlike the Look-Up Table block, the Look-Up Table Dynamic block allows you to change the table data without stopping the simulation. For example, you may want to automatically incorporate new table data if the physical system you are simulating changes.

You define the look-up table by inputting the `x` and `y` table data to the block as 1-by-n vectors. To help reduce the ROM used by the code generated for this block, you can use different data types for the `x` table data and the `y` table data. However, these restrictions apply:

- The `y` table data and the output vector must have the same sign, the same bias, and the same fractional slope.
- The `x` table data and the `x` data vector must have the same sign, the same bias, and the same fractional slope. Additionally, the precision and range for the `x` data vector must greater than or equal to the precision and range for the `x` table data.

The block generates output based on the input values using one of these methods selected from the **Look-up method** parameter list:

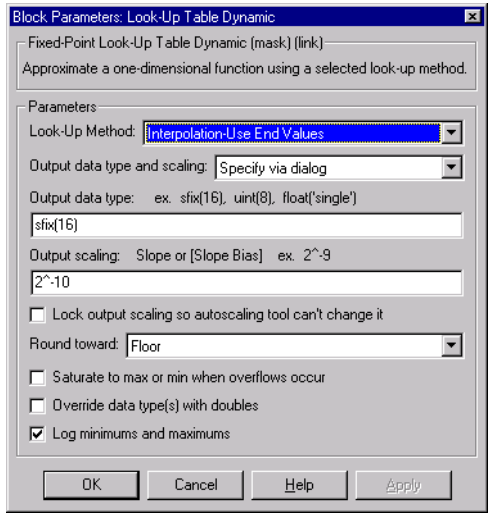
- `Interpolation-Extrapolation` – This is the default method; it performs linear interpolation and extrapolation of the inputs.

  - If a value matches the block's input, the output is the corresponding element in the output vector.
  - If no value matches the block's input, then the block performs linear interpolation between the two appropriate elements of the table to determine an output value. If the block input is less than the first or greater than the last input vector element, then the block extrapolates using the first two or last two points.

- `Interpolation-Use End Values` – This method performs linear interpolation as described above but does not extrapolate outside the end points of the input vector. Instead, the end-point values are used.
- `Use Input Nearest` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest the current input is found. The corresponding element in `y` is then used as the output.
- `Use Input Below` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest and below the current input is found. The corresponding element in `y` is then used as the output. If there is no element in `x` below the current input, then the nearest element is found.
- `Use Input Above` – This method does not interpolate or extrapolate. Instead, the element in `x` nearest and above the current input is found. The corresponding element in `y` is then used as the output. If there is no element in `x` above the current input, then the nearest element is found.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Parameters and Dialog Box



Look-up method

Look-up method.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override with data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Conversions

The table data is converted from doubles to the x data type. This conversion is performed offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

Example

For an example that illustrates the look-up methods supported by this block, see the example included in the Look-Up Table block reference pages.

Characteristics

Input Port(s)	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	No

See Also

Look-Up Table, and related blocks

Look-Up Table (2D)

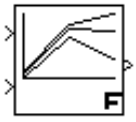
Purpose

Approximate a two-dimensional function using a selected look-up method

Library

LookUp

Description



The Look-Up Table (2-D) block is a masked S-function that computes an approximation to some function `z=f(x,y)` given `x`, `y`, `z` data points. The look-up method can use interpolation, extrapolation, or the original values of the inputs. Also, the `x` and `y` data vectors must be strictly monotonically increasing as described in the Look-Up Table reference pages.

The **Row** parameter is a 1-by-m vector of `x` data points, the **Col** parameter is a 1-by-n vector of `y` data points, and the **Table** parameter is an m-by-n matrix of `z` data points. Both the row and column vectors must be strictly monotonically increasing. The block generates output based on the input values using one of these methods selected from the **Method** parameter list:

- `Interpolation-Extrapolation` – This is the default method; it performs linear interpolation and extrapolation of the inputs.
  - If the inputs match row and column parameter values, the output is the value at the intersection of the row and column.
  - If the inputs do not match row and column parameter values, then the block generates output by linearly interpolating between the appropriate row and column values. If either or both block inputs are less than the first or greater than the last row or column values, the block extrapolates from the first two or last two points.
- `Interpolation-Use End Values` – This method performs linear interpolation as described above but does not extrapolate outside the end points of the input vector. Instead, the end-point values are used.
- `Use Input Nearest` – This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest the current inputs are found. The corresponding element in `z` is then used as the output.
- `Use Input Below` – This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest and below the current inputs are found. The corresponding element in `z` is then used as the output. If there are no elements in `x` or `y` below the current inputs, then the nearest elements are found.

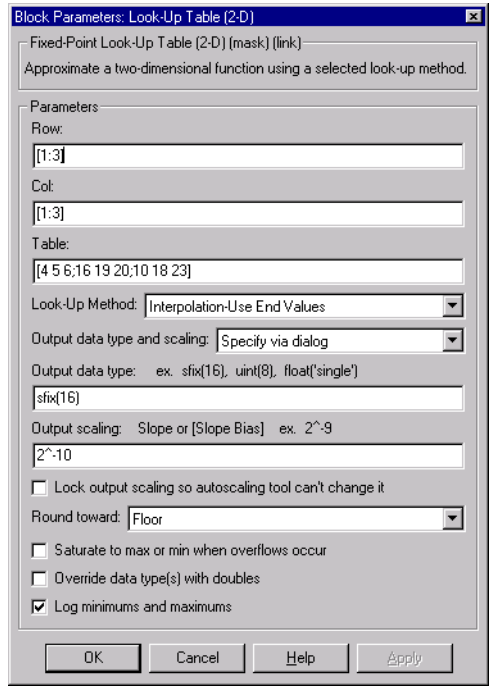
- `Use Input Above` – This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest and above the current inputs are found. The corresponding element in `z` is then used as the output. If there are no elements in `x` or `y` above the current inputs, then the nearest elements are found.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Remarks

To avoid parameter saturation errors, the automatic scaling script `autofixexp` employs a special rule for the Look-Up Table (2D) block. `autofixexp` modifies the scaling by using the output look-up values in addition to the logged minimum and maximum simulation values. This prevents the data from being saturated to different values. The look-up values are given by the **Table** parameter (the `TableDataPoints` variable).

Parameters and Dialog Box



Row

Input row vector. It must be strictly monotonically increasing.

Col

Input column vector. It must be strictly monotonically increasing.

Table

Output vector. It must match the size defined by the **Row** and **Col** parameters.

Look-up method

Look-up method.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it.

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

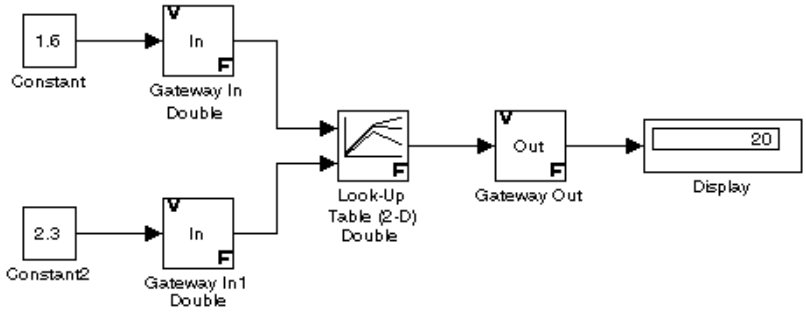
If checked, minimum and maximum simulation values are logged to the workspace.

Conversions

The **Row** parameter is converted from doubles to the first input's data type. The **Col** parameter is converted from doubles to the second input's data type. The **Table** parameter is converted from doubles to the output data type. All conversion are performed offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

Example

Suppose the Look-Up Table (2D) block is configured to use input row and column vectors given by `[1:3]`, and a look-up table given by `[4 5 6; 16 19 20; 10 18 23]`. Using the model shown below,



the following results were generated.

Look-Up Method	Input [x y]	Output	Comment
`Interpolation-Extrapolation`	`[1.6 2.5]`	13.9	N/A
	`[1.6 4.0]`	15.4	N/A
`Interpolation-Use End Values`	`[1.6 2.5]`	13.9	N/A
	`[1.6 4.0]`	14.4	The value for `[1.6 3]` was used.
`Use Input Nearest`	`[1.6 2.3]`	19	The value for `[2 2]` was used.

Look-Up Method	Input [x y]	Output	Comment
Use Input Below	[1.6 2.3]	5	The value for [1 2] was used.
	[1.6 0.5]	4	The value for [1 1] was used.
Use Input Above	[1.6 2.3]	20	The value for [2 3] was used.
	[1.6 3.5]	20	The value for [2 3] was used.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of one input if the other is a vector

See Also

Look-Up Table, and related blocks

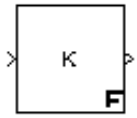
Purpose

Multiply the input by a constant matrix

Library

Math

Description



The Matrix Gain block is a masked S-function that multiplies the input by a constant matrix (referred to as the matrix gain). The block generates its output by multiplying the input by a specified matrix

$$y = Ku$$

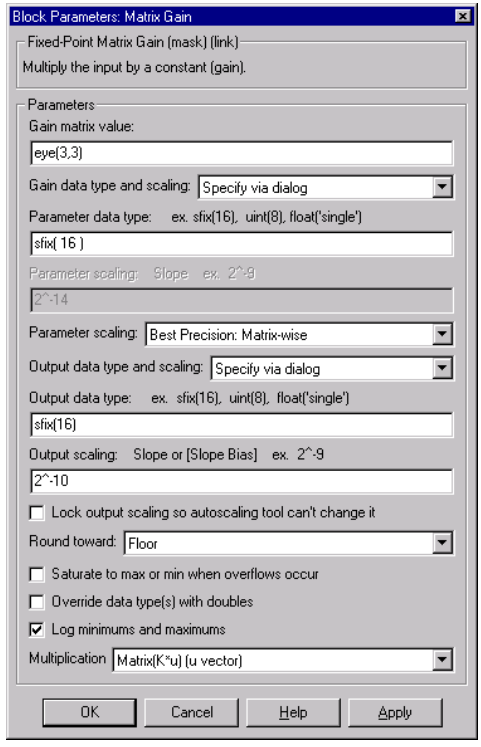
where K is the matrix gain and u is the input. If the matrix has m rows and n columns, then the input to this block should be a vector of length n. The output is a vector of length m.

You specify the matrix gain with the **Gain matrix value** parameter. You specify the scaling for the matrix gain with the **Parameter scaling** parameter. Note that there are two dialog box parameters that control the matrix gain scaling: one associated with an edit field, and one associated with a parameter list. If **Parameter data type** is a generalized fixed-point number such as `sfix(16)`, the **Parameter scaling** list provides you with these scaling modes:

- `Use Specified Scaling` – This mode uses the slope/bias or radix point-only scaling specified for the editable **Parameter scaling** parameter (for example, `2^-10`).
- Use `Best Precision: Element-wise` – This mode produces radix points such that the precision is maximized for each element of the **Gain matrix value** matrix.
- Use `Best Precision: Row-wise` – This mode produces a common radix point for each element of a **Gain matrix value** row based on the best precision for the largest value of that row.
- Use `Best Precision: Column-wise` – This mode produces a common radix point for each element of a **Gain matrix value** column based on the best precision for the largest value of that column.
- Use `Best Precision: Matrix-wise` – This mode produces a common radix point for each element of the **Gain matrix value** matrix based on the best precision for the largest value of the matrix.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

Parameters and Dialog Box



Gain matrix value

Specify as a scalar or vector.

Gain data type and scaling

Set the data type and scaling of the gain.

Parameter data type

Any data type supported by the Fixed-Point Blockset.

Parameter scaling

Radix point-only or slope/bias scaling. Additionally, the gain can be scaled using the constant matrix scaling modes for maximizing precision. These scaling modes are available only for generalized fixed-point data types.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data types(s) with doubles

If checked, the **Parameter data type** and **Output data type** values are overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Conversions and Operations

The **Gain matrix value** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

The Matrix Gain block first multiples its inputs by the **Gain matrix value** parameter, converts those results to the output data type using the specified rounding and overflow modes, and then performs the summation. Refer to "Rules for Arithmetic Operations" on page 4-30 for more information about the rules this block adheres to when performing operations.

Matrix Gain

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	No

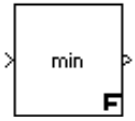
Purpose

Output the minimum or maximum input value

Library

Math

Description



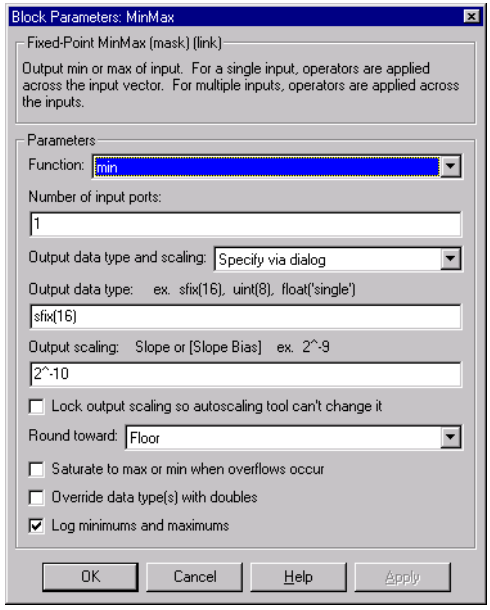
The MinMax block is a masked S-function that outputs either the minimum or the maximum element of the inputs. You can choose which function to apply with the **Function** parameter list.

You specify the number of input ports with the **Number of input ports** parameter. If the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

If the block has multiple input ports, the non-scalar inputs must all have the same dimensions. The block expands any scalar inputs to have the same dimensions as the non-scalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

Parameters and Dialog Box



Function

The function to apply to the input.

Number of input ports

The number of inputs to the block.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

MinMax Running Resettable

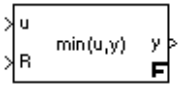
Purpose

Determine the minimum or maximum of a signal over time

Library

Math

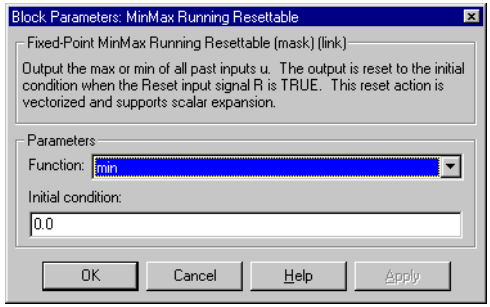
Description



The MinMax Running Resettable block outputs the minimum or maximum of all past inputs u. You specify whether the block outputs the minimum or the maximum with the **Function** parameter.

The block can reset its state based on an external reset signal R. When the reset signal R is TRUE, the block resets the output to the value of the **Initial condition** parameter.

Parameters and Dialog Box



Function

Specify whether the block outputs the minimum or the maximum.

Initial condition

Initial condition.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as the input u
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

MinMax

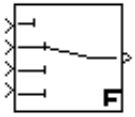
Purpose

Switch output between different inputs based on the value of the first input

Library

Select

Description



The MultiPort Switch block is a masked S-function that passes through the data input specified by the first (top) input. The first input is called the *control input*, while the rest of the inputs are called *data inputs*.

If the control input is an integer value, then the specified data input is passed through to the output. For example, if the control input is 1, then the first data input is passed through to the output. If the control input is not an integer value, the block truncates the value to an integer by rounding to floor. If the truncated control input is less than 1 or greater than the number of input ports, an out-of-bounds error is returned.

The block inputs can be scalar or vector. You specify the number of data inputs with the **Number of input ports** parameter. The block output is determined by these rules:

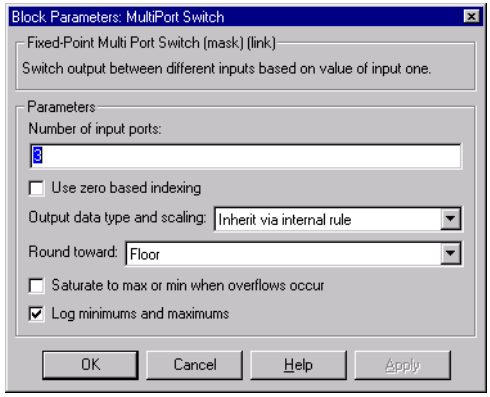
- If you specify only one data input and that input is a vector, the block behaves as an "index selector," and not as a multi-port switch. The block output is the vector element that corresponds to the value of the control input.
- If you specify more than one data input, the block behaves like a multi-port switch. The block output is the data input that corresponds to the value of the control input. If at least one of the data inputs is a vector, the block output is a vector. Any scalar inputs are expanded to vectors.
- If the inputs are scalar, the output is a scalar.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Note The output data type is determined by the data input with the largest positive range.

The Index Vector block, also in the Select library, is another implementation of the MultiPort Switch block having different parameter settings.

Parameters and Dialog Box



Number of input ports

The number of data inputs to the block.

Use zero based indexing

Controls whether the first item has index zero or index one.

Output data type and scaling

Inherit the output data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output. This parameter does not apply to an integer control input.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap. This parameter does not apply to an integer control input.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset

Direct Feedthrough	Yes
Scalar Expansion	Yes

Product

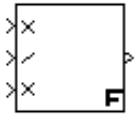
Purpose

Multiply or divide inputs

Library

Math

Description



The Product block is a masked S-function that performs multiplication or division of its inputs.

You specify the operations with the **Enter */ characters or the number of inputs** parameter. Multiply-divide characters indicate which operations are to be performed:

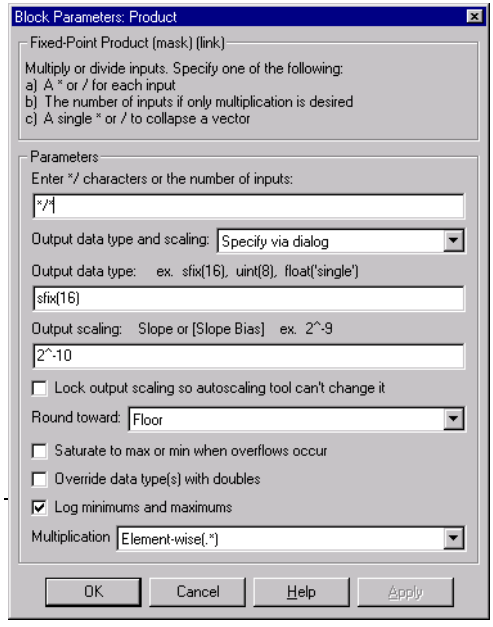
- If there are two or more inputs, then the number of multiply-divide characters must equal the number of inputs. For example, `'*/*'` requires three inputs and configures the block to divide the first (top) input by the second (middle) input, and then and multiply the third (bottom) input. If the first character is `'/'`, then the first input is inverted.
- If only multiplication of inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of multiply-divide characters.
- If only one vector is input, then a single `'*'` or `'/'` will collapse the vector using the specified operation.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

The Math library contains the following implementations, which are all linked to the Product block but have different parameter settings:

- Multiply
- Divide
- Product of Elements
- Product of Elements Inverted
- Multiply Matrix

Parameters and Dialog Box



Enter */ characters or the number of inputs

Enter as many multiply or divide characters as there are inputs. For multiplication only, you can enter the number of inputs since this is the default operation.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Operations

The Product block first performs the specified multiply or divide operations on the inputs, and then converts the results to the output data type using the specified rounding and overflow modes. Refer to “Rules for Arithmetic Operations” on page 4-30 for more information about the rules this block adheres to when performing operations.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

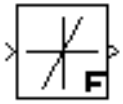
Purpose

Limit the rising and falling rates of the signal

Library

Nonlinear

Description

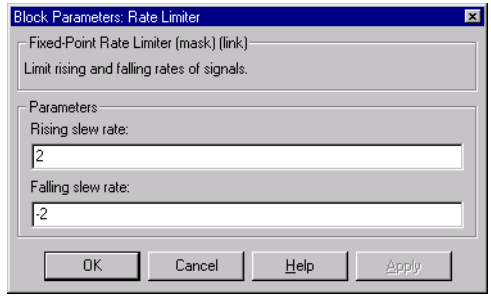


The Rate Limiter block is a masked block that limits the rising and falling rates of the signal.

Use the **Rising slew rate** parameter to set the limit on the rising rate of the signal.

Use the **Falling slew rate** parameter to set the limit on the falling rate of the signal.

Parameters and Dialog Box



Rising slew rate

Limit on the rising rate of the signal.

Falling slew rate

Limit on the falling rate of the signal.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

Rate Limiter Dynamic

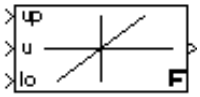
Purpose

Limit the rising and falling rates of the signal

Library

Nonlinear

Description

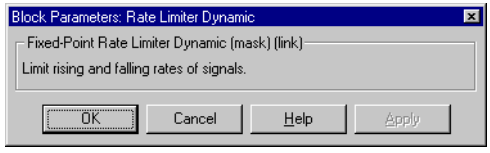


The Rate Limiter Dynamic block is a masked block that limits the rising and falling rates of the signal.

The external signal up sets the upper limit on the rising rate of the signal.

The external signal lo sets the lower limit on the falling rate of the signal.

Parameters and Dialog Box



Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same data type as input u
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Rate Limiter

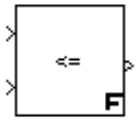
Purpose

Perform the specified relational operation on the inputs

Library

Logic & Comparison

Description



The Relational Operator block is a masked S-function that performs a comparison of its two inputs. The first (top) input is converted to the data type of the second (bottom) input prior to comparison.

The operator connecting the two inputs is selected with the **Operator** parameter list. The supported relational operators are given below.

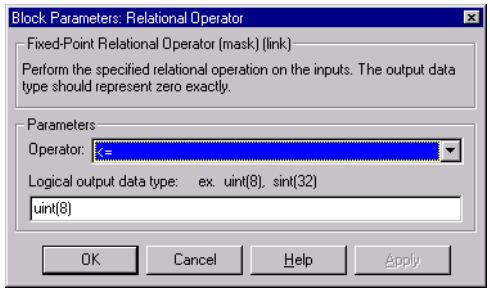
Operation	Description
==	TRUE if the first input is equal to the second input
~=	TRUE if the first input is not equal to the second input
<	TRUE if the first input is less than the second input
<=	TRUE if the first input is less than or equal to the second input
>=	TRUE if the first input is greater than or equal to the second input
>	TRUE if the first input is greater than the second input

The output is specified with the **Logical output data type** parameter. The output equals 1 for TRUE and 0 for FALSE.

Note The output data type selected should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

Relational Operator

Parameters and Dialog Box



Operator

Relational operator used to compare the two inputs.

Logical output data type

Output data type. You should only use data types that can represent zero exactly.

Conversions

The input with the smallest positive range is converted to the data type of the other input offline using round-to-nearest and saturation. This conversion is performed prior to comparison. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Any data type supported by the blockset that can exactly represent zero
Direct Feedthrough	Yes
Scalar Expansion	Of inputs

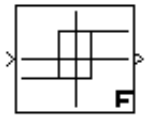
Purpose

Switch output between two constants

Library

Nonlinear

Description



The Relay block is a masked S-function that allows the output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

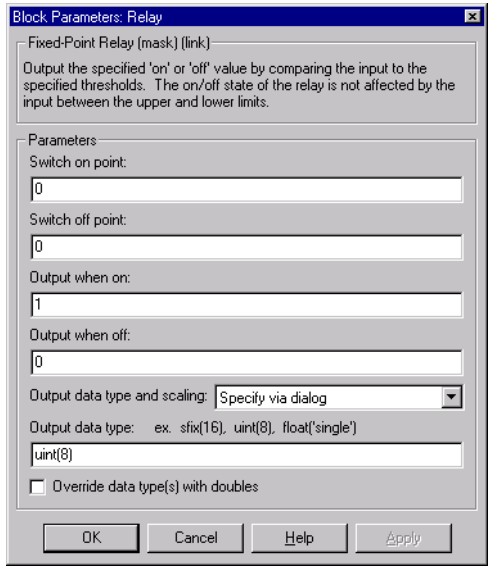
The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** value models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

You specify the output scaling with the **Output scaling** parameter. Note that there are two dialog box parameters that control the output scaling: one associated with an edit field, and one associated with a parameter list. If **Output data type** is a generalized fixed-point number such as `sfix(16)`, the **Output scaling** parameter list provides you with these scaling modes:

- `Use Specified Scaling` – This mode uses the slope/bias or radix point-only scaling specified for the editable **Output scaling** parameter (for example, `2^-10`).
- `Best Precision: Vector-wise` – This mode produces the best precision based on the **Output when on** and **Output when off** parameters.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

Parameters and Dialog Box



Switch on point

The “on” threshold for the relay.

Switch off point

The “off” threshold for the relay.

Output when on

The output when the relay is on.

Output when off

The output when the relay is off.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset. If you specify `ufix`, the **Output Scaling** field appears in the dialog box. If you then select **Use Specified Scaling** in the **Output Scaling** field, you can enter the scaling for the output in the **Output Scaling: Slope or [Slope Bias]** field.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Conversions

Both the **Switch on point** and **Switch off point** parameters are converted to the input data type offline using round-to-nearest and saturation.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

Repeating Sequence Interpolated

Purpose

Output discrete-time sequence and repeat, interpolating between data points

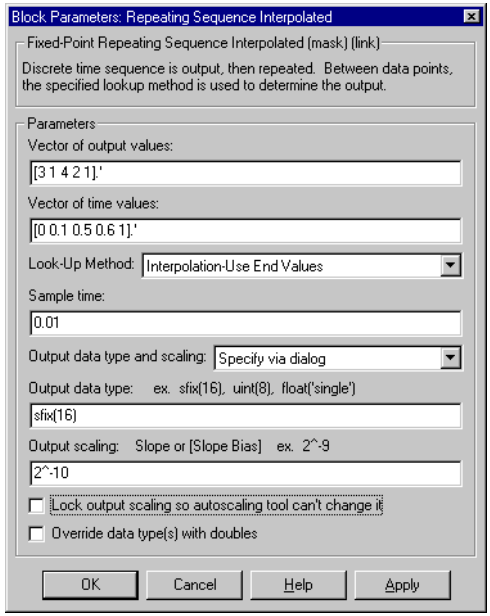
Library

Sources

Description

The Repeating Sequence Interpolated block outputs a discrete-time sequence and then repeats it. Between data points, the block uses the method specified by the **Look-up Method** parameter to determine the output.

Parameters and Dialog Box



Vector of output values

Column vector containing output values of the discrete time sequence.

Vector of time values

Column vector containing time values. The time values must be a strictly increasing and the vector must have the same size as the vector of output values.

Look-up Method

Specify the lookup method to determine the output between data points.

Sample time

Sample time.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or by inheriting the data type and scaling by back propagation.

Output data type

Any data type supported by the blockset.

Output scaling

Select the scaling method using the specified scaling or using the best precision.

Lock output scaling so autoscaling tool can't change it

If the box is checked, output scaling is locked.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Characteristics

Output Port	Any data type supported by the blockset
Scalar Expansion	Yes

Repeating Sequence Stair

Purpose

Output and repeat the discrete time sequence

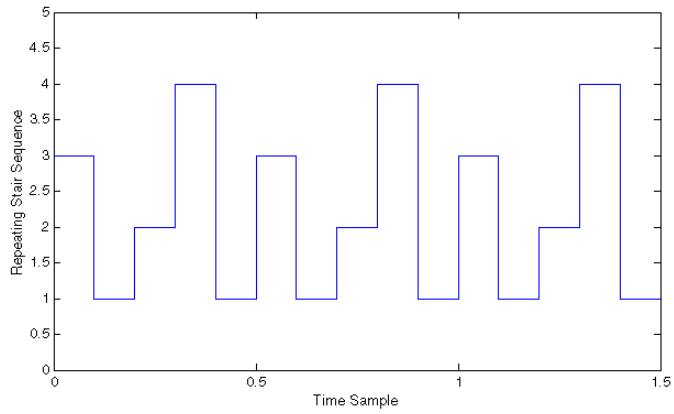
Library

Sources

Description

The Repeating Sequence Stair block is a masked block that outputs and repeats a discrete time sequence.

You can specify the stair sequence with the **Vector of output values** parameter. For example, the vector can be specified as `[3 1 2 4 1]'`, producing the stair sequence shown in the plot.



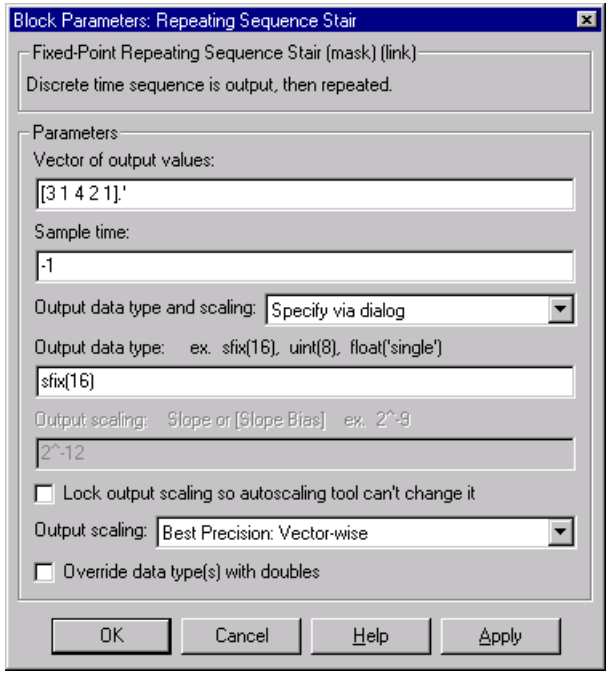
You can specify the sample time with the **Sample time** parameter.

You can select the output data type and scaling with the **Output data type and scaling** parameter, and set the output data type with the **Output data type** parameter.

For fixed-point data types, you can set the output scaling with the **Output scaling** parameter, and, below that parameter, select the method for scaling the output with the **Output scaling** parameter.

For a detailed description of all block parameters, refer to “Block Parameters” on page 9-4. For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-26.

Parameters and Dialog Box



Vector of output values

Vector containing values of the repeating stair sequence.

Sample time

Sample time.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or by inheriting the data type and scaling by back propagation.

Output data type

Any data type supported by the blockset.

Output scaling

Slope or slope/bias scaling.

Lock output scaling so autoscaling tool can't change it

If the box is checked, output scaling is locked.

Output scaling

Select the scaling method using the specified scaling or using the best precision.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Characteristics

Output Port	Any data type supported by the blockset
Scalar Expansion	No
Vectorized	No

See Also

Repeating Sequence Interpolated

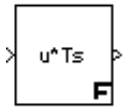
Purpose

Support calculations involving sample time

Library

Calculus

Description



The Sample Time Multiply block is a masked S-function that adds, subtracts, multiplies, or divides the input signal, `u`, by a weighted sample time `Ts`.

You specify the math operation with the **Operation** parameter. Additionally, you can specify to use only the weight with either the sample time or its inverse.

Enter the weighting factor with the **Weight value**. If the weight is 1, `w` is removed from the equation.

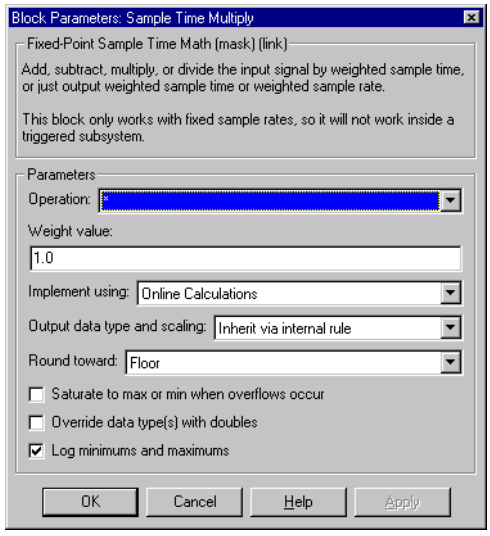
For a detailed description of all block parameters, refer to “Block Parameters” on page 9-4. For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-26.

The Calculus library contains the following implementations, which are all linked to the Sample Time Multiply block but have different parameter settings:

- Sample Time Divide
- Sample Time Add
- Sample Time Subtract
- Sample Time Probe
- Sample Rate Probe

Sample Time Multiply

Parameters and Dialog Box



Operation

Specify operation to use: `+`, `-`, `*`, `/`, `Ts` only, `1/Ts` only.

Weight value

Enter weight of sample time.

Implement using

Specify online calculations or offline scaling adjustment.

Output data type and scaling

Specify whether the output data type and scaling are inherited by an internal rule or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the output data type is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workshops.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	For all math operations options except `Ts` and `1/Ts`
Scalar Expansion	No, the weight is always a scalar

Saturation

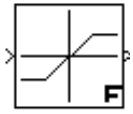
Purpose

Bound the range of the input

Library

Nonlinear

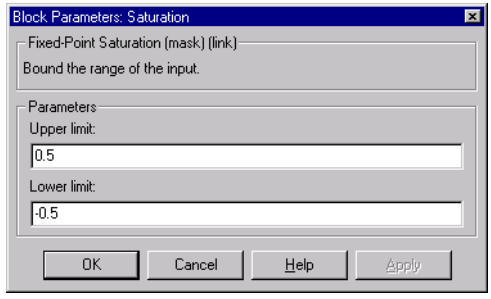
Description



The Saturation block is a masked S-function that limits the input signal to upper and lower saturation values.

You specify the upper bound of the input with the **Upper limit** parameter and the lower bound of the input with the **Lower limit** parameter. If the input signal is outside these limits, the output saturates to one of the bounds.

Parameters and Dialog Box



Upper limit

The upper bound on the input signal.

Lower limit

The lower bound on the input signal.

Conversions

Both the **Upper limit** and **Lower limit** parameters are converted to the input data type offline using round-to-nearest and saturation.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as input data type
Direct Feedthrough	Yes
Scalar Expansion	Of input and limits

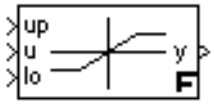
Purpose

Bound the range of the input

Library

Nonlinear

Description

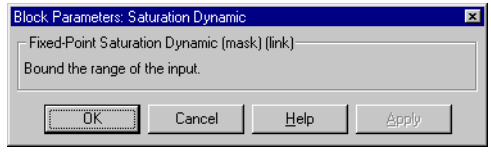


The Saturation Dynamic block is a masked block that bounds the range of the input signal to upper and lower saturation values. The input signal outside of these limits saturates to one of the bounds where:

- The input below the lower limit is set to the lower limit.
- The input above the upper limit is set to the upper limit.

The input for the upper limit is the `up` port, and the input for the lower limit is the `lo` port.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

Saturation

Scaling Strip

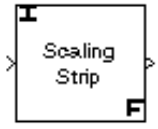
Purpose

Remove scaling and map to a built in integer

Library

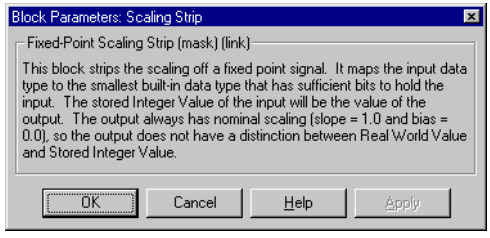
Data Type

Description



The Scaling Strip block strips the scaling off a fixed point signal. It maps the input data type to the smallest built in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not make a distinction between real world value and stored integer value.

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

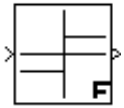
Purpose

Indicate the sign of the input

Library

Nonlinear

Description

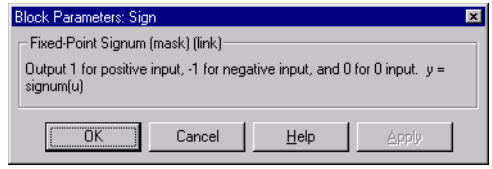


The Sign block is a masked S-function that indicates the sign of the input:

- The output is 1 when the input is greater than zero.
- The output is 0 when the input is equal to zero.
- The output is -1 when the input is less than zero.

The output is a signed data type with the same number of bits as the input, and with nominal scaling (a slope of one and a bias of zero).

Parameters and Dialog Box



Characteristics

Input Port	Any data type supported by the blockset
Output Port	A signed Fixed-Point Blockset data type
Direct Feedthrough	Yes
Scalar Expansion	N/A

Sine

Purpose

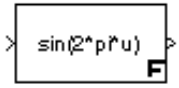
Implement a sine wave in fixed-point using a lookup table approach that exploits quarter wave symmetry

Library

Lookup

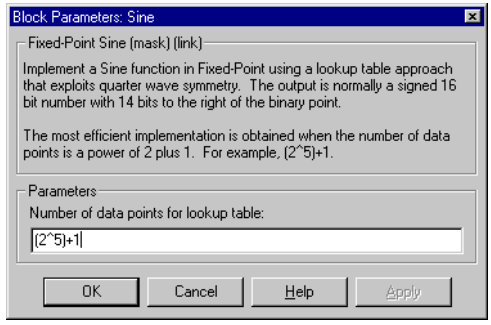
Description

The Sine block is a masked block that implements a sine wave in fixed-point using a lookup table method that exploits quarter wave symmetry.



You can set the number of data points to retrieve from the lookup table with the **Number of data points for lookup table** parameter.

Parameters and Dialog Box



Number of data points for lookup table

Number of data points to retrieve from the lookup table.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	N/A

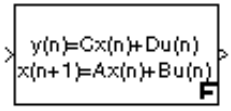
Purpose

Implement discrete-time state space

Library

Filters

Description



The State-Space block implements the system described by

$$y(n) = Cx(n) + Du(n)$$

$$x(n+1) = Ax(n) + Bu(n)$$

where u is the input, x is the state, and y is the output. Both equations have the same data type.

The matrices A, B, C and D have the following characteristics:

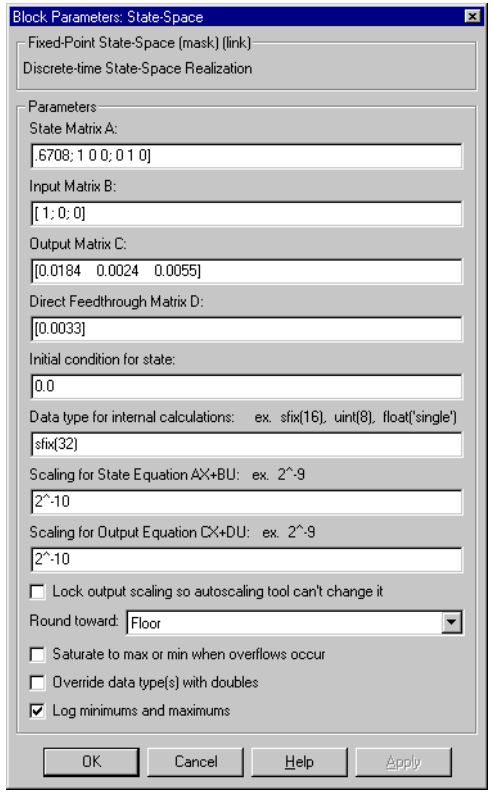
- A must be an n-by-n matrix, where n is the number of states.
- B must be an n-by-m matrix, where m is the number of inputs.
- C must be an r-by-n matrix, where r is the number of outputs.
- D must be an r-by-m matrix.

In addition:

- The state x must be a n-by-1 vector
- The input u must be a m-by-1 vector
- The output y must be a r-by-1 vector

The block accepts one input and generates one output. The input vector width is determined by the number of columns in the B and D matrices. The output vector width is determined by the number of rows in the C and D matrices.

Parameters and Dialog Box



State Matrix A

Matrix of states.

Input Matrix B

Column vector of inputs.

Output Matrix C

Column vector of outputs.

Direct Feedthrough Matrix D

Matrix for direct feedthrough.

Initial condition for state

Initial condition for the state.

Data type for internal calculations

Data type for internal calculations. Some examples are `sfix(16)`, `unit(8)`, and `float('single')`.

Scaling for State Equation AX+BU

Scaling for state equations.

Scaling for Output Equation CX+DU

Scaling for output equations.

Lock output scaling so autoscaling tool can't change it

If the box is checked, the output scaling is locked.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Characteristics

Input Ports	Any data type supported by the blockset – it must be a scalar
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Of initial conditions
Vectorized	No

Sum

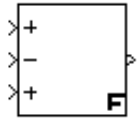
Purpose

Add or subtract inputs

Library

Math

Description



The Sum block is a masked S-function that performs addition or subtraction on its inputs.

You specify the operations with the **Enter +- characters or the number of inputs** parameter. Plus-minus characters indicate the operations to be performed on the inputs:

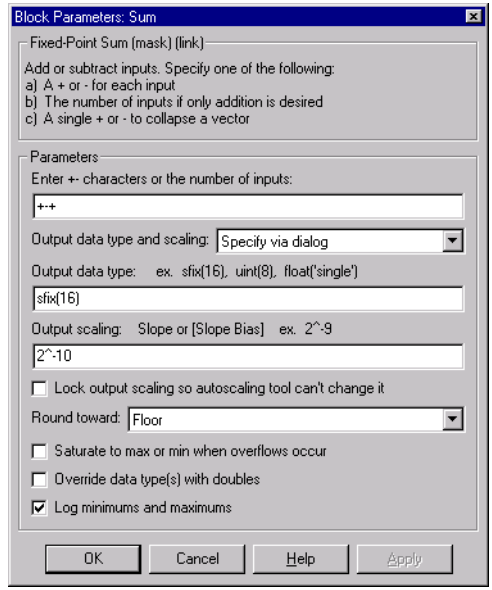
- If there are two or more inputs, then the number of plus-minus characters must equal the number of inputs. For example, `'+-+'` requires three inputs and configures the block to subtract the second (middle) input from the first (top) input, and then add the third (bottom) input.
- If only addition of inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of plus-minus characters.
- If only one vector is input, then a single `'+'` or `'-'` will collapse the vector using the specified operation.

For a detailed description of all other block parameters, refer to “Block Parameters” on page 10-4.

The Math Library contains the following implementations, which are all linked to the Sum block but have different parameter settings:

- Add
- Subtract
- Sum of Elements
- Sum of Elements Negated

Parameters and Dialog Box



Enter +- characters or the number of inputs

Enter as many plus or minus characters as there are inputs. For addition only, you can enter the number of inputs since this is the default operation.

Output data type and scaling

Specify the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back propagation.

Output data type

Any data type supported by the Fixed-Point Blockset.

Output scaling

Radix point-only or slope/bias scaling. These scaling modes are available only for generalized fixed-point data types.

Lock output scaling so autoscaling tool can't change it

If checked, **Output scaling** is locked. This feature is available only for generalized fixed-point output.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Override data type(s) with doubles

If checked, the **Output data type** is overridden with doubles.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Operations

The Sum block first converts the input data type(s) to the output data type using the specified rounding and overflow modes, and then performs the specified operations. Refer to “Rules for Arithmetic Operations” on page 4-30 for more information about the rules this block adheres to when performing operations.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Any data type supported by the blockset
Direct Feedthrough	Yes
Scalar Expansion	Yes

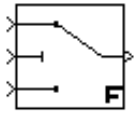
Purpose

Switch output between the first input and the third input based on the value of the second input

Library

Select

Description



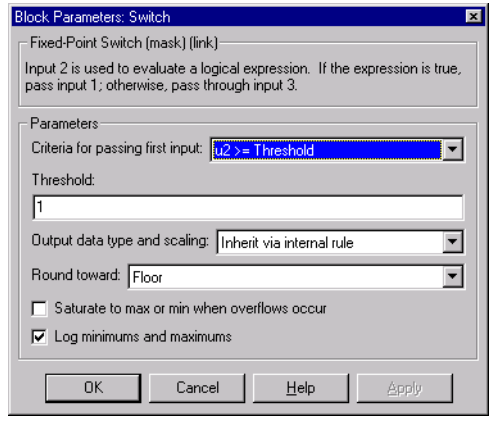
The Switch block is a masked S-function that passes through the first (top) input or the third (bottom) input based on the value of the second (middle) input. The second input is called the *control input*.

The first input is passed through when the second input is greater than or equal to the value of the **Threshold** parameter. Otherwise, it passes the third input through. The threshold value is converted to the second input's data type.

For a detailed description of all other block parameters, refer to "Block Parameters" on page 10-4.

Note The output data type is determined by the input with the largest positive range. If the first input has a larger positive range than the third input, then it specifies the output data type. Otherwise, the third input specifies the output data type.

Parameters and Dialog Box



Threshold

Switch threshold that determines which input is passed to the output.

Output data type and scaling

Inherit the output data type and scaling from the driving block or by back propagation.

Round toward

Rounding mode for the fixed-point output.

Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Log minimums and maximums

If checked, minimum and maximum simulation values are logged to the workspace.

Conversions

The **Threshold** parameter is converted offline to the second input's data type using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-27 for more information about parameter conversions.

Characteristics

Input Ports	Any data type supported by the blockset
Output Port	Same as input port one
Direct Feedthrough	Yes
Scalar Expansion	Yes

See Also

MultiPort Switch

Purpose

Delay a scalar signal multiple sample periods and output all the delayed versions

Library

Delays & Holds

Description

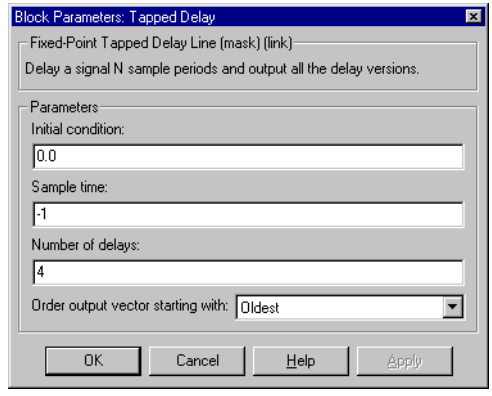
The Tapped Delay block delays its input by the specified number of sample periods, and outputs all the delayed versions.

This block provides a mechanism for discretizing a signal in time, or resampling the signal at a different rate. You specify the time between samples with the **Sample time** parameter. You specify the number of delays with the **Number of delays** parameter. A value of `-1` instructs the block to inherit the number of delays by back propagation. Each delay is equivalent to the z^{-1} discrete-time operator, which is represented by the Unit Delay block.

The block accepts one scalar input and generates an output for each delay. The input must be a scalar. You specify the order of the output vector with the **Order output vector starting with** parameter list. **Oldest** orders the output vector starting with the oldest delay version and ending with the newest delay version. **Newest** orders the output vector starting with the newest delay version and ending with the oldest delay version.

The block output for the first sampling period is specified by the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior.

Parameters and Dialog Box



Initial condition

The initial output of the simulation.

Sample time

Sample time.

Number of delays

The number of discrete-time operators.

Order output vector starting with

Specify whether the oldest delay version is output first, or the newest delay version is output first.

Conversions

The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	No
Scalar Expansion	Yes – of initial conditions

Purpose

Negate the input

Library

Math

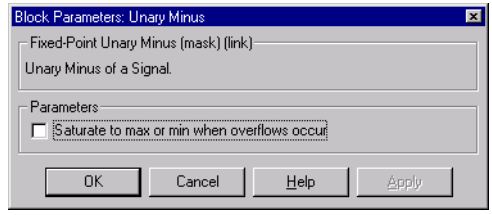
Description

The Unary Minus block is a masked S-function that negates the input. The block accepts only signed data types.

For signed data types, you cannot accurately negate the most negative value since the result is not representable by the data type. In this case, the behavior of the block is controlled by the **Saturate to max or min when overflows occur** check box. If checked, the most negative value of the data type wraps to the most positive value. If not checked, the operation has no effect. If an overflow occurs, then a warning is returned to the MATLAB command line.

For example, suppose the block input is an 8-bit signed integer. The range of this data type is from -128 to 127, and the negation of -128 is not representable. If the **Saturate to max or min when overflows occur** check box is checked, then the negation of -128 is 127. If it is not checked, then the negation of -128 remains at -128.

Parameters and Dialog Box



Saturate to max or min when overflows occur

If checked, fixed-point overflows saturate. Otherwise, they wrap.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input (a nonzero bias is negated offline)
Direct Feedthrough	No
Scalar Expansion	Of input or initial conditions

Unit Delay

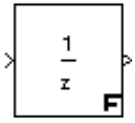
Purpose

Delay a signal one sample period

Library

Delays & Holds

Description



The Unit Delay block is a masked S-function that delays its input by the specified sample period. This block is equivalent to the z^{-1} discrete-time operator. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are delayed by the same sample period.

You specify the block output for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior. The time between samples is specified with the **Sample time** parameter. A setting of -1 means the **Sample time** is inherited.

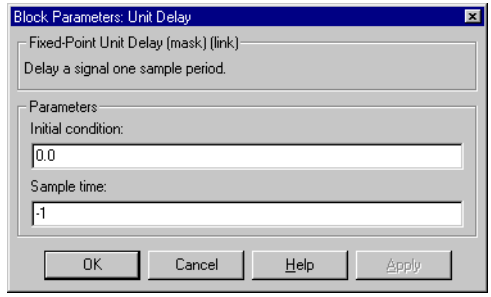
Note The Unit Delay block accepts continuous sample times. When it has a continuous sample time, the block is equivalent to the built-in Memory block.

Remarks

This block provides a mechanism for discretizing one or more signals in time, or resampling the signal at a different rate. If your model contains multirate transitions, then you must add Unit Delay blocks between the slow to fast transitions. The sample rate of the Unit Delay must be set to that of the slower block.

For fast to slow transitions, use the Zero Order Hold block. For more information about multirate transitions, refer to *Using Simulink* or the *Real-Time Workshop User's Guide*.

Parameters and Dialog Box



Initial condition

The initial output of the simulation.

Sample time

Sample time.

Conversions

The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	No
Scalar Expansion	Of input or initial conditions

Unit Delay Enabled

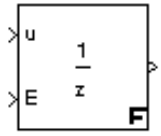
Purpose

Delay a signal one sample period

Library

Delays & Holds

Description



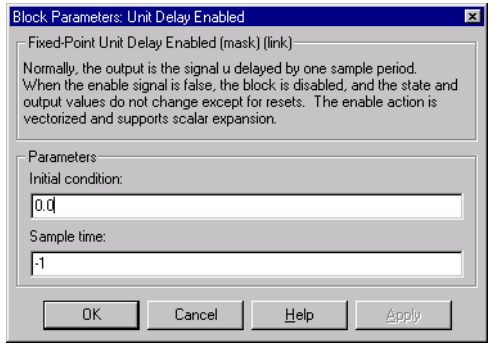
The Unit Delay Enabled block is a masked block that delays a signal by one sample period when the external enable signal E is on. While the enable is off, the block is disabled. It holds the current state at the same value and outputs that value. The enable signal is on when E is not 0, and off when E is 0.

You specify the block output for the first sampling period with the value **Initial condition** parameter.

The output data type is the same as the input u data type. The data type of the input u and the enable E can be any data type.

You input the sample time with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset

Output Port	Same as the input u
Direct Feedthrough	No
Scalar Expansion	Yes

See Also Unit Delay, and related blocks

Unit Delay Enabled External IC

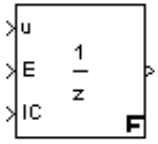
Purpose

Delay a signal one sample period, if the enable is on

Library

Delays & Holds

Description



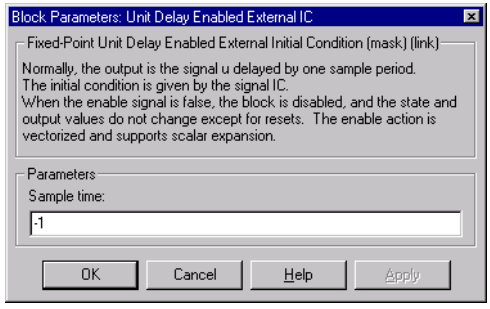
The Unit Delay Enabled block is a masked block that delays a signal by one sample period when the enable signal E is on. While the enable is off, the block holds the current state at the same value and outputs that value. The enable E is on when E is not 0, and off when E is 0.

The blocks initial condition is given by the signal IC.

The input u and `IC` data types must be the same, and are any data type. The output data type is the same as u and `IC`. The enable E is any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Input Port IC	Same as the input u
Output Port	Same as the input u

Direct Feedthrough	Yes, of the reset input port No, of the enable input port Yes, of the external IC port
Scalar Expansion	Yes

See Also Unit Delay, and related blocks

Unit Delay Enabled Resettable

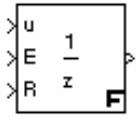
Purpose

Delay a signal one sample period

Library

Delays & Holds

Description



The Unit Delay Enabled Resettable block combines the features of the Unit Delay Enabled and Unit Delay Resettable blocks.

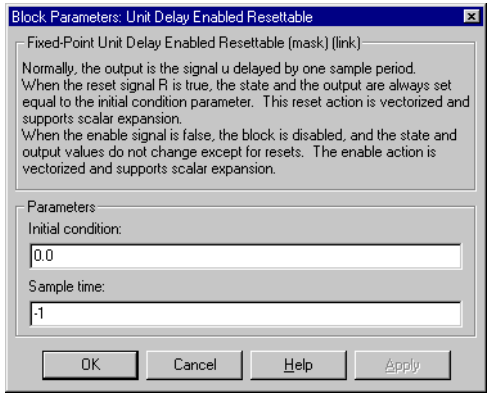
The block can reset its state based on an external reset signal R. When the enable signal E is on and the reset signal R is false, the block outputs the input signal delayed by one sample period.

When the enable signal E is on and the reset signal R is true, the block resets the current state to the initial condition, specified by the **Initial condition** parameter, and outputs that state delayed by one sample period.

When the enable signal is off, the block is disabled, and the state and output do not change except for resets. The enable signal is on when E is not 0, and off when E is 0.

You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Initial Condition

The initial output of the simulation.

Sample Time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Output Port	Same as the input u
Direct Feedthrough	No, of the input port No, of the enable port Yes, of the reset port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Unit Delay Enabled Resettable External IC

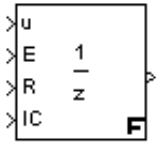
Purpose

Delay a signal one sample period, if the enable is on

Library

Delays & Holds

Description



The Unit Delay Enabled Resettable External IC block combines the features of the Unit Delay Enabled, Unit Delay External IC, and Unit Delay Resettable blocks.

The block can reset its state based on an external reset signal R. When the enable signal E is on and the reset signal R is false, the block outputs the input signal delayed by one sample period.

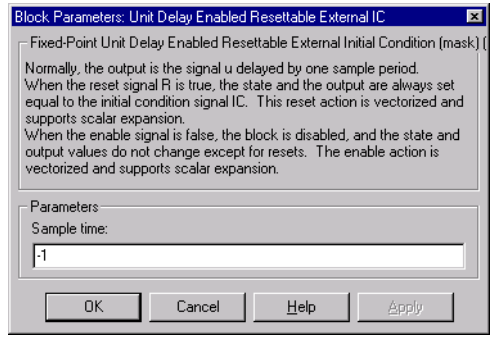
When the enable signal E is on and the reset signal R is true, the block resets the current state to the initial condition given by the signal IC, and outputs that state delayed by one sample period.

When the enable signal is off, the block is disabled, and the state and output do not change except for resets. The enable signal is on when E is not 0, and off when E is 0.

The output data type is the same as the input u and the initial condition IC data type, which can be any data type, but must be the same. The enable E and reset R can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Input Port IC	Same as the input u
Output Port	Same as the input u
Direct Feedthrough	No, of the input port No, of the enable port Yes, of the enable port Yes, of the external IC port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Unit Delay External IC

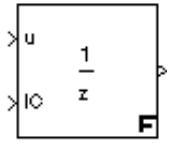
Purpose

Delay a signal one sample period

Library

Delays & Holds

Description



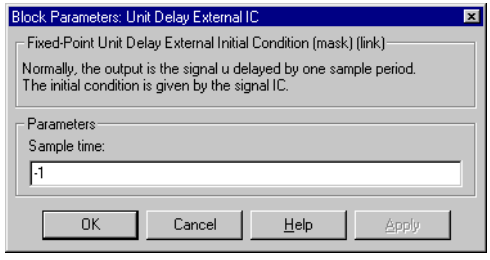
The Unit Delay block is a masked block that delays its input by one sample period. This block is equivalent to the z^{-1} discrete-time operator. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are delayed by the same sample period.

The block's output for the first sample period is equal to the signal IC.

The input u and initial condition `IC` data types must be the same, and are any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port IC	Same as the input u
Output Port	Same as the input u
Direct Feedthrough	No, of the input port Yes, of the external IC port
Scalar Expansion	Yes

Purpose

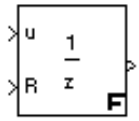
Delay a signal one sample period

Library

Delays & Holds

Description

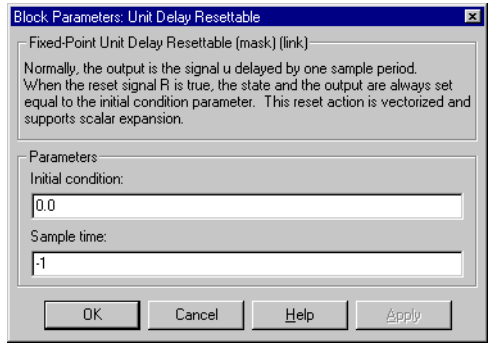
The Unit Delay Resettable block delays a signal one sample period.



The block can reset its state based on an external reset signal R. The block has two input ports, one for the input signal u and the other for the external reset signal R. When the reset signal is false, the block outputs the input signal delayed by one time step. When the reset signal is true, the block resets the current state to the initial condition, specified by the **Initial condition** parameter, and outputs that state delayed by one time step.

You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Parameters and Dialog Box



Initial Condition

The initial output of the simulation.

Sample Time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Output Port	Same as the input u

Direct Feedthrough	No, of the input port Yes, of the reset port
Scalar Expansion	Yes

See Also Unit Delay, and related blocks

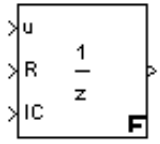
Purpose

Delay a signal one sample period, with reset and external initial condition

Library

Delays & Holds

Description



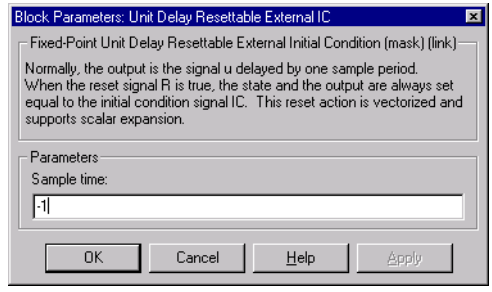
The Unit Delay Resettable External IC block delays a signal one sample period.

The block can reset its state based on an external reset signal R. The block has two input ports, one for the input signal u and the other for the reset signal R. When the reset signal is false, the block outputs the input signal delayed by one time step. When the reset signal is true, the block resets the current state to the initial condition given by the signal IC and outputs that state delayed by one time step.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Parameters and Dialog Box



Sample Time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Input Port IC	Same as the input u
Output Port	Same as the input u

Unit Delay Resettable External IC

Direct Feedthrough	No, of the input port Yes, of the reset port Yes, of the external IC port
Sample Time	Inherited
Scalar Expansion	Yes

See Also Unit Delay, and related blocks

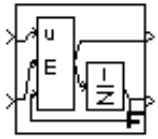
Purpose

Support calculations that have feedback and depend on the current input

Library

Delays & Holds

Description



The Unit Delay With Preview Enabled block supports calculations that have feedback and depend on the current input.

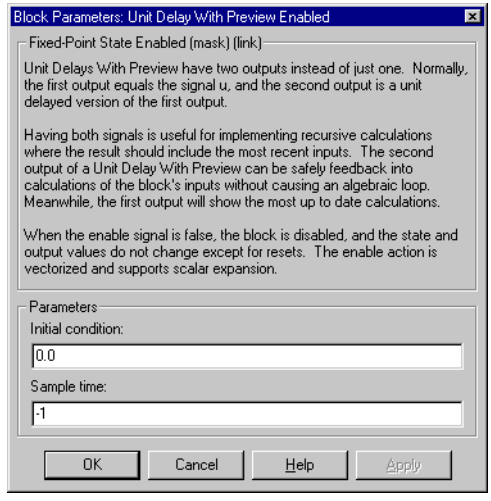
The block has two output ports. When the external enable signal E is on, the upper port outputs the signal and the lower port outputs the signal delayed by one sample period. The block has two input ports, one for the input signal u and the other for the enable signal E.

When the enable signal E is off, the block is disabled, and the state and output values do not change, except for resets. The enable signal is on when E is not 0, and off when E is 0.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Parameters and Dialog Box



Unit Delay With Preview Enabled

Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Output Ports	Same as the input u
Direct Feedthrough	Yes, to upper output port No, to lower output port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Unit Delay With Preview Enabled Resettable

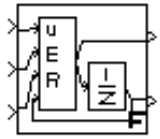
Purpose

Support calculations that have feedback and depend on the current input

Library

Delays & Holds

Description



The Unit Delay With Preview Enabled Resettable block supports calculations that have feedback and depend on the current input.

The block can reset its state based on an external reset signal R. The block has two output ports. When the external enable signal E is on and the reset R is false, the upper port outputs the signal and the lower port outputs the signal delayed by one sample period. The block has two input ports, one for the input signal u and the other for the enable signal E.

When the enable signal E is on and the reset R is true, the block resets the current state to the initial condition given by the **Initial condition** parameter. The block outputs that state delayed by one sample time through the lower output port, and outputs the state without a delay through the upper output port.

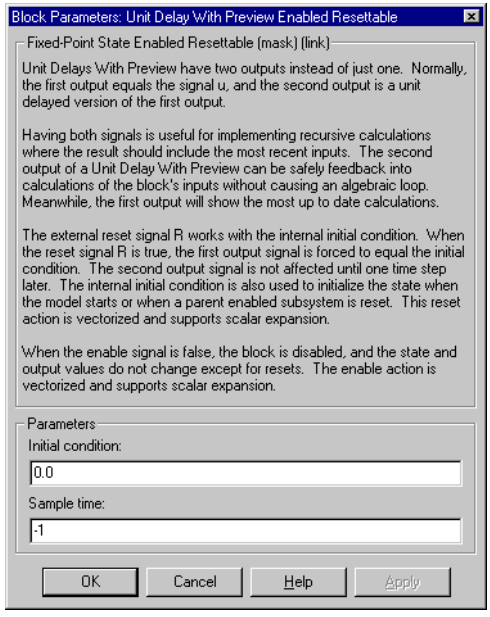
When the Enable signal is off, the block is disabled, and the state and output values do not change, except for resets. The enable signal is on when E is not 0, and off when E is 0.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Unit Delay With Preview Enabled Resettable

Parameters and Dialog Box



Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Output Ports	Same as the input u
Direct Feedthrough	Yes, to upper output port No, to lower output port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Unit Delay With Preview Enabled Resettable External RV

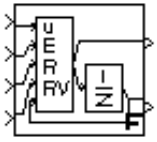
Purpose

Support calculations that have feedback and depend on the current input

Library

Delays & Holds

Description



The Unit Delay With Preview Enabled Resettable External RV block supports calculations that have feedback and depend on the current input.

The block can reset its state based on an external reset signal R. The block has two output ports. When the external enable signal E is on and the reset R is false, the upper port outputs the signal and the lower port outputs the signal delayed by one sample period. The block has two input ports, one for the input signal u and the other for the enable signal E.

When the enable signal E is on and the reset R is true, the upper output signal is forced to equal the external reset signal RV. The lower output signal is not affected until one time step later, at which time it is equal to the external reset signal RV at the previous time step. The block uses the internal **Initial condition** only when the model starts or when a parent enabled subsystem is used. The internal **Initial condition** only affects the lower output signal. The first output is only affected through feedback.

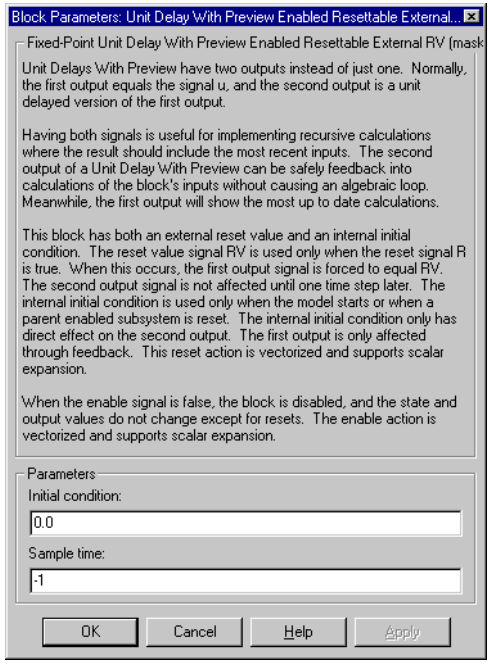
When the Enable signal is off, the block is disabled, and the state and output values do not change, except for resets. The enable signal is on when E is not 0, and off when E is 0.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Unit Delay With Preview Enabled Resettable External RV

Parameters and Dialog Box



Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port E	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Input Port RV	Same as the input u
Output Ports	Same as the input u

Direct Feedthrough	Yes, to upper output port No, to lower output port
Scalar Expansion	Yes

See Also Unit Delay, and related blocks

Unit Delay With Preview Resettable

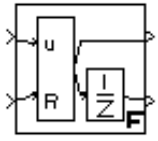
Purpose

Support calculations that have feedback and depend on the current input

Library

Delays & Holds

Description



The Unit Delay With Preview Resettable block supports calculations that have feedback and depend on the current input.

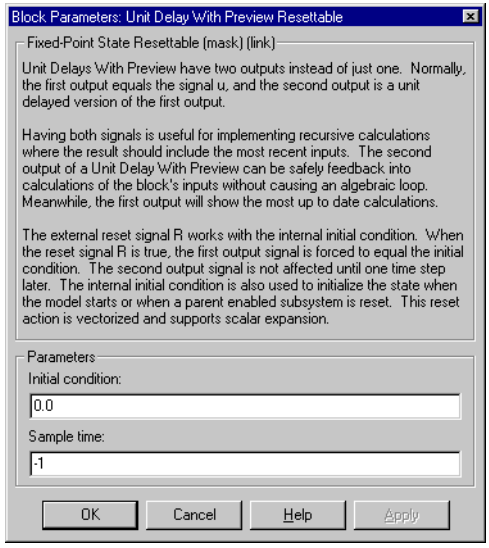
The block can reset its state based on an external reset signal R. The block has two output ports. When the reset R is false, the upper port outputs the signal and the lower port outputs the signal delayed by one sample period.

When the reset R is true, the block resets the current state to the initial condition given by the **Initial condition** parameter. The block outputs that state delayed by one sample time through the lower output port, and outputs the state without a delay through the upper output port.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Parameters and Dialog Box



Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Output Ports	Same as the input u
Direct Feedthrough	Yes, to upper output port No, to lower output port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Unit Delay With Preview Resettable External RV

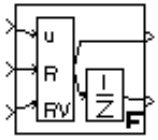
Purpose

Support calculations that have feedback and depend on the current input

Library

Delays & Holds

Description



The Unit Delay With Preview Resettable External RV block supports calculations that have feedback and depend on the current input.

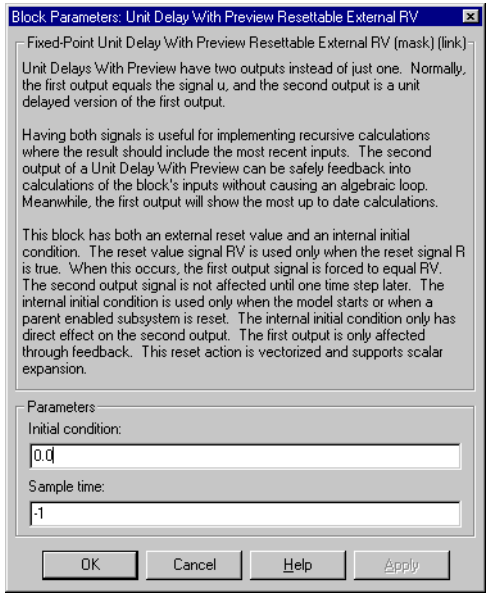
The block can reset its state based on an external reset signal R. The block has two output ports. When the external reset R is false, the upper port outputs the signal and the lower port outputs the signal delayed by one sample period.

When the external reset R is true, the upper output signal is forced to equal the external reset signal RV. The lower output signal is not affected until one time step later, at which time it is equal to the external reset signal RV at the previous time step. The block uses the internal **Initial condition** only when the model starts or when a parent enabled subsystem is used. The internal **Initial condition** only affects the lower output signal. The first output is only affected through feedback.

The input `u` and initial condition `IC` must be the same data type, but can be any data type. The output is the same data type as the inputs `u` and `IC`. The reset `R` can be any data type.

You specify the time between samples with the **Sample time** parameter**.** A setting of `-1` means the **Sample time** is inherited.

Parameters and Dialog Box



Initial condition

Initial condition.

Sample time

Sample time.

Characteristics

Input Port u	Any data type supported by the blockset
Input Port R	Any data type supported by the blockset
Input Port RV	Same as the input u
Output Ports	Same as the input u
Direct Feedthrough	Yes, to upper output port No, to lower output port
Scalar Expansion	Yes

See Also

Unit Delay, and related blocks

Wrap To Zero

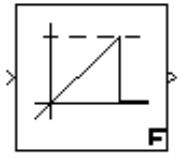
Purpose

Set output to zero if input is above threshold

Library

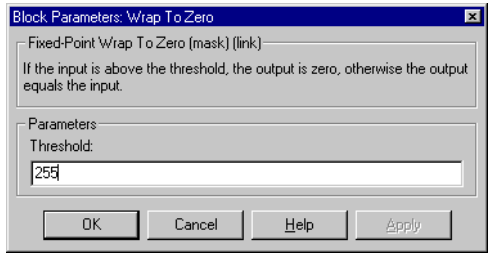
Nonlinear

Description



The Wrap To Zero block sets the output to zero if the input is above the value set by the **Threshold** parameter, and outputs the input if the input is less than or equal to the **Threshold**.

Parameters and Dialog Box



Threshold

When the input exceeds the threshold, the output is set to zero.

Characteristics

Input Port	Any data type supported by the blockset
Output Ports	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	Yes

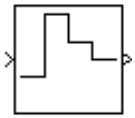
Purpose

Implement a zero-order hold of one sample period

Library

Delays & Holds

Description



The Zero-Order Hold block is a masked S-function that samples and holds its input for the specified sample period. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are held for the same sample period.

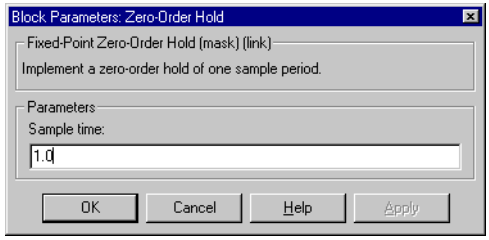
You specify the time between samples with the **Sample time** parameter**.** A setting of -1 means the **Sample time** is inherited.

Remarks

This block provides a mechanism for discretizing one or more signals in time, or resampling the signal at a different rate. If your model contains multirate transitions, you must add Zero-Order Hold blocks between the fast to slow transitions. The sample rate of the Zero-Order Hold must be set to that of the slower block.

For slow to fast transitions, use the Unit Delay block. For more information about multirate transitions, refer to *Using Simulink* or the *Real-Time Workshop User's Guide*.

Parameters and Dialog Box



Sample time

Sample time.

Characteristics

Input Port	Any data type supported by the blockset
Output Port	Same as the input
Direct Feedthrough	Yes
Scalar Expansion	No

A

Code Generation

Overview A-2

Code Generation Support A-3
Languages A-3
Storage Class of Variables A-3
Storage Class of Parameters A-3
Rounding Modes A-3
Overflow Handling A-4
Blocks A-4
Scaling A-4

Generating Pure Integer Code A-5
Example: Generating Pure Integer Code A-5

Using the Simulink Accelerator A-11

Using External Mode or rsim Target A-12
External Mode A-12
Rapid Simulation Target A-12

Customizing Generated Code A-13
Macros Versus Functions A-13
Bit Sizes for Target C Compiler A-13

Overview

With the Real-Time Workshop, the Fixed-Point Blockset can generate C code. The code generated from fixed-point blocks uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations. You can use the generated code on embedded fixed-point processors or rapid prototyping systems even if they contain a floating-point processor. The code is structured so that key operations can be readily replaced by optimized target-specific libraries that you supply. You can also use the Target Language Compiler to customize the generated code. For more information about code generation, refer to the *Real-Time Workshop User's Guide* and the *Target Language Compiler Reference Guide*.

You can also generate code for testing on a rapid prototyping system such as xPC, the Real-Time Windows Target, or dSPACE. The target compiler and processor may support floating-point operations in software or in hardware. In any case, the fixed-point blocks will generate pure integer code and will not use floating-point operations. This allows valid bit-true testing even on a floating-point processor.

You can also generate code for non real-time testing. For example, code can be generated to run in non real-time on computers running any supported operating system. Even though the processors have floating-point hardware, the code generated by fixed-point blocks is pure integer code. The Generic Real-Time Target (GRT) and the Simulink Accelerator are examples of where non real-time code is generated and run.

Code Generation Support

All fixed-point blocks support code generation, but not every simulation feature is supported. The code generation support is described below.

Languages

- C support only
- No Ada support

Storage Class of Variables

- Fixed-Point Blockset code generation handles variables that do not match the target compiler's sizes for char, short, int, or long. Code generation supports any variable having a width less than or equal to a long, either signed or unsigned. For example, the C40 compiler defines a long to be 32 bits. Therefore, the allowable sizes for variables range between 1 and 32 bits. This capability is particularly useful if you want to:
  - Prototype on one target chip, but use a different target chip for production.
  - Provide bit-true simulation in a rapid prototyping environment for odd data type sizes used by FPGA's, ASIC's, 24-bit DSP's, and so on.
- No floating-point support except for the fixed-point gateway blocks.

Storage Class of Parameters

- The Real-Time Workshop external mode support requires that parameters be 1 to 32 bits, either signed or unsigned. The parameter size must also be compatible with the target C compiler.
- No floating-point support

Rounding Modes

- All four rounding modes are supported.
- Rounding to floor generates the most efficient code for most cases.

Overflow Handling

- Saturation mode is supported.
- Wrapping mode is supported and generates the most efficient code.
- Automatic exclusion of saturation code when hardware saturation is available is currently not supported. Wrapping must be selected for the Real-Time Workshop to exclude saturation code.

Blocks

All blocks generate code for all operations with a few exceptions:

- The Look-Up Table, Look-Up Table (2D), and Dynamic Look-Up Table blocks generate code for all look-up methods except extrapolation.
- A few combinations of scaling and operations lead to highly inefficient code. These few cases are described in the next section.

Scaling

- Radix point-only scaling is supported.
- Slope/bias scaling is supported for all blocks except when it leads to highly inefficient code. All blocks except four support all cases of slope/bias scaling. The Gain, Matrix Gain, and FIR blocks support matched slope/bias scaling where the block input signals and output signals have the same slopes and biases, but not mismatched slope/bias scaling. The Product block supports mismatched slope, but not mismatched bias. For more information about matched and mismatched slope/bias scaling, refer to “Signal Conversions” on page 4-27.

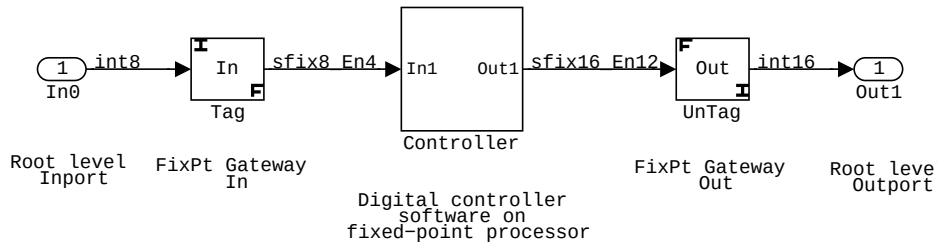
  It is generally recommended that signals with slope/bias scaling (such as a sensor input) are immediately converted to radix point-only scaling. This will typically produce more efficient code.

Generating Pure Integer Code

All blocks generate pure integer code except for the Gateway In, Gateway In Inherited, and Gateway Out blocks. These blocks must generate floating-point code when handling floating-point input or output. However, if the input or output is an integer and the block is configured to treat the input or output as a stored integer, then these blocks will also generate pure integer code.

Example: Generating Pure Integer Code

This example outlines the steps you should take when generating pure integer code for your Fixed-Point Blockset model. The steps follow the description in the `fxpdemo_code_only` demo, which includes the model shown below.



Note This example generates code using the Embedded C Real-Time Target (ERT), whch is available with the RTW Production Coder. If your version of the Real-Time Workshop does not support ERT code generation, then you may want to select the Generic Real-Time Target (GRT). Using GRT, all Fixed-Point Blockset blocks (except the gateway blocks) will generate pure integer code. However, the code related to the GRT infrastructure is not generated to exclude floating-point operations. For example, GRT may decide when to execute blocks based on a floating-point counter.

---

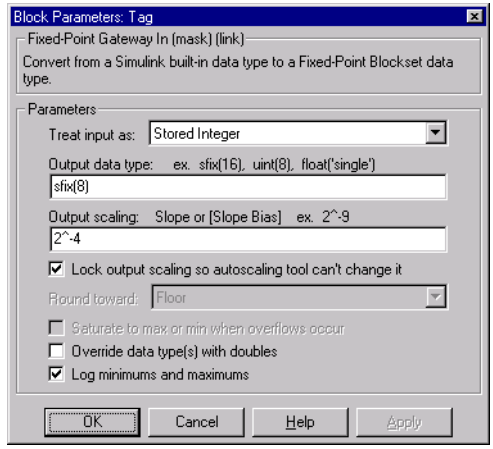
1 Copy the fixed-point portion of your model to a new model.

If your original model includes blocks that represent hardware, analog systems, and other blocks not related to embedded software, then you must

create a new model. This new model contains only the fixed-point portion, which represents the software that will be running on the fixed-point processor. For example, the digital controller subsystem shown above contains the fixed-point blocks from the `fxpdemo_feedback` model used for code generation.

2 Insert Gateway In blocks, as needed.

- Change the **Treat input as:** parameter from `Real World Value` to `Stored Integer`. This does not change the signal's value, but it is needed to "tag" integers with fixed point scaling information. The Gateway In block dialog box for this configuration is shown below.

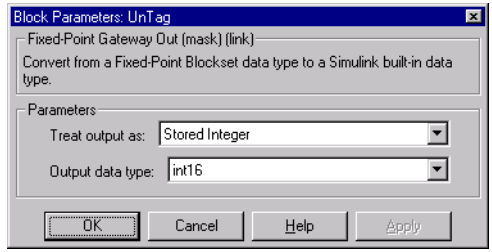


- Precede all Gateway In blocks with root level Inport blocks, and configure the blocks to use the appropriate integer data type. For example, the Inport block shown above is configured to use the built-in `int8` data type.

3 Insert Gateway Out Blocks, as needed.

- Change the **Treat input as:** parameter from `Real World Value` to `Stored Integer`. This does not change the signal's value, but it is needed to "strip" fixed-point scaling information from the integer. Also, configure the

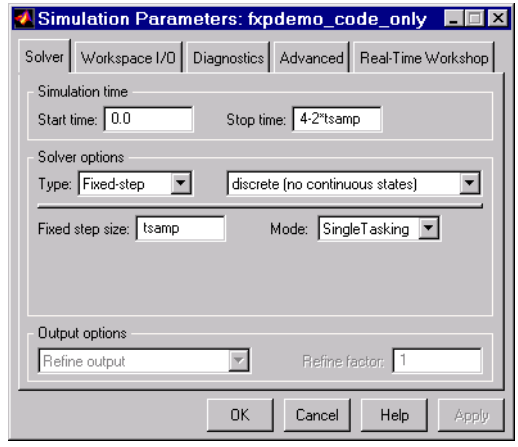
Output date type: parameter to use the appropriate integer data type. The Gateway Out block dialog box for this configuration is shown below.



- Follow all Gateway Out blocks with root level Outport blocks.

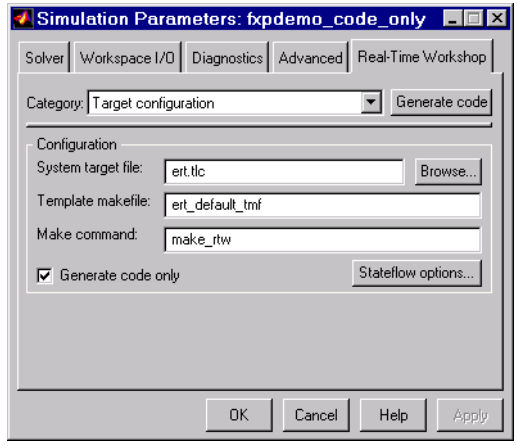
4 Configure the simulation parameters.

- Launch Simulink's Simulation Parameter's dialog box by selecting **Parameters** under the **Simulation** menu.
- In the **Solver** window, configure **Solver options:** to `Fixed-step` and `discrete (no continuous states)`, and configure **Fixed step size:** to the required value. The **Solver** window for this configuration is shown below.



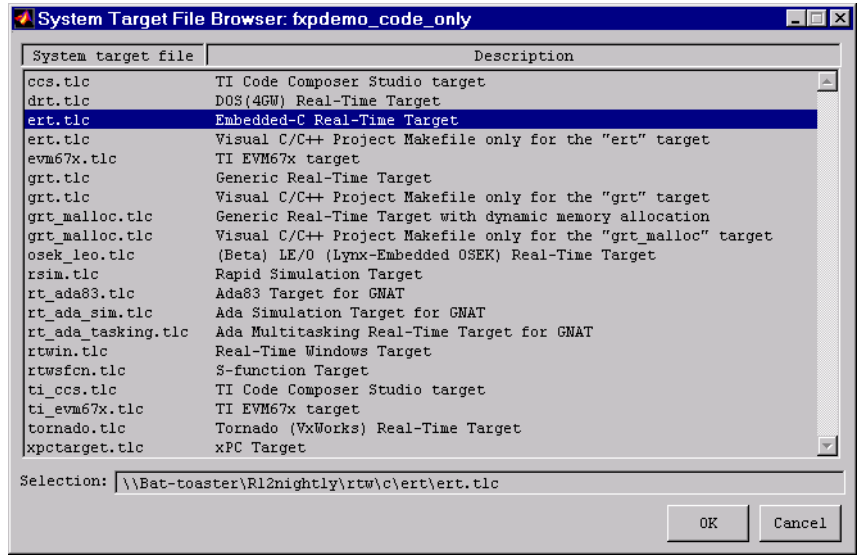
- Select the **Real-Time Workshop** window in the Simulation Parameters dialog box. Configure the **System target file** parameter to `ert.tlc`. The

Template makefile and **Make command** parameters are automatically updated. This configuration is shown below.



Launch the System Target File Browser by selecting the **Browse** button in the **Configuration** panel. If it is available, select Embedded-C Real-Time

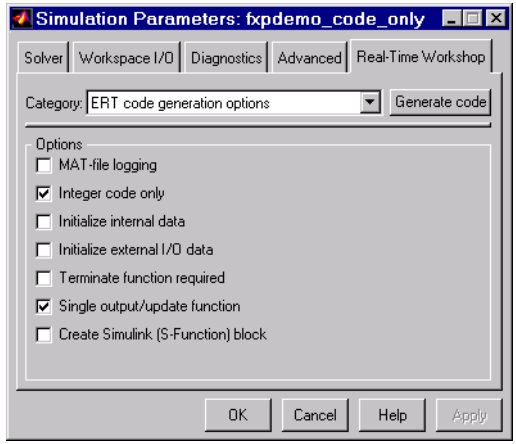
Target as the system target file and hit the **OK** button. The System Target File Browser for this configuration is shown below.



The Fixed-Point Blockset supports all targets except those that generate Ada code. Note that you may not have ERT code generation capability. If this is the case, you should select the Generic Real-Time Target.

- To configure the code generation parameters, select **ERT code generation options** from the **Category** parameter list. Select the **Integer code only** check box and any other options you may require. To configure additional code generation optimizations such as inlining, select **General code generation options** from the **Category** parameter list. The ERT code

generation options for this configuration are shown below. If you are using GRT, the dialog box choices are slightly different.



Note that all fixed-point blocks except gateway blocks produce pure integer code for all supported targets.

- Build the code by selecting the **Generate code** button.

Using the Simulink Accelerator

You can use the Simulink Accelerator with your Fixed-Point Blockset model if the model meets the code generation restrictions.

The Simulink Accelerator can drastically increase the speed of some fixed-point models. This is especially true for models that execute at a very large number of time steps. The time overhead to generate code for a fixed-point model will generally be larger than the time overhead to set up a model for simulation. As the number of time steps increases, the relative importance of this overhead decreases.

Refer to the *Using Simulink* guide for more information about the Simulink Accelerator.

Using External Mode or rsim Target

If you are using the Real-Time Workshop external mode or rsim (rapid simulation) target, there are situations where you may get unexpected errors when tuning block parameters.

These errors can arise when you use blocks that support constant scaling for best precision and you use the "best precision" scaling option. To avoid these errors, you should use the `Use Specified Scaling` parameter value. Refer to “Example: Constant Scaling for Best Precision” on page 3-12 for a description of the constant scaling feature. Refer to Chapter 10, “Block Reference” for a description of blocks that support this feature.

For more information about external mode or rapid simulation target, refer to the *Real-Time Workshop User's Guide*.

External Mode

If you change a fixed-point block parameter by a sufficient amount (approximately a factor of two), the radix point changes. If you change a parameter such that the radix point moves during an external mode simulation (or during graphical editing) and you reconnect to the target, a checksum error occurs and you must rebuild the code.

For example, suppose a block has a parameter value of `-2`. You then build the code and connect in external mode. While connected, you change the parameter to `-4`. If the simulation is stopped and then restarted, this parameter change causes a radix point change. In external mode, the radix point is kept fixed. If you keep the parameter value of `-4` and disconnect from the target, then when you reconnect, a checksum error occurs and you must rebuild the code.

Rapid Simulation Target

If a parameter change is great enough, and you are using the best precision mode for constant scaling, then you cannot use the rapid simulation target.

If you change a block parameter by a sufficient amount (approximately a factor of two), the best precision mode changes the radix point. Any change in the radix point requires the code to be rebuilt since the model checksum is changed. This means that if best precision parameters are changed over a great enough range, you cannot use the rapid simulation target and a checksum error message occurs when you initialize the rsim executable.

Customizing Generated Code

You can customize generated code by directly modifying the TLC file `fixpttarget.tlc`, which is located in the `fixpoint` directory. The two most important customizations are described below.

Macros Versus Functions

You can modify the TLC file to generate macros or C functions calls. With macros, you can avoid the overhead of a function call. With function calls, you can significantly reduce the overall code size for large routines. Additionally, many debuggers will not allow you to single-step through macros. This is not the case with function calls. The factory default setting is to generate macros.

Bit Sizes for Target C Compiler

You can modify the TLC file to accommodate custom target sizes by explicitly specifying the number of bits defined for char, short, int, or long data types.

If you do not manually override these sizes, then the sizes for the MATLAB host computer are automatically selected. For example, if you are running MATLAB under the Windows operating system, then char, short, int, and long default to 8, 16, 32, and 32 bits, respectively. Most other supported operating systems use the same data type sizes. However, DEC Alpha for example, defines a long as 64 bits.

B

Selected Bibliography

1 Burrus, C. S., J.H. McClellan, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

2 Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems, Second Edition*; Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

3 *Handbook For Digital Signal Processing*, edited by S.K. Mitra and J.F. Kaiser; John Wiley & Sons, Inc., New York, 1993.

4 Hanselmann, H., “Implementation of Digital Controllers — A Survey”; *Automatica*, vol. 23, no. 1, pp 7-32, 1987.

5 Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*; Kluwer Academic Publishers, Seventh Printing, Norwell, Massachusetts, 1993.

6 Middleton, R. and G. Goodwin, *Digital Control and Estimation – A Unified Approach*; Prentice Hall, Englewood Clifs, New Jersey. 1990.

7 Moler, C., "Floating points: IEEE Standard unifies arithmetic model", Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at `http://www.mathworks.com/company/newsletter/clevescorner/cleve_toc.shtml`

8 Ogata, K., *Discrete-Time Control Systems, Second Edition*; Prentice Hall, Englewood Cliffs, New Jersey, 1995.

9 Roberts, R.A. and C.T. Mullis, *Digital Signal Processing*; Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

Index

A

accumulation
  scaling recommendations 4-19
  using slope/bias encoding 4-19
accumulator data type 7-3
  and feedback controller demo 6-8
addition 10-192
  blockset rules 4-30
  scaling recommendations 4-17
  using slope/bias encoding 4-16
ALU's 4-30
arithmetic shift 4-42
`autofixexp` 9-4
automatic scaling
  and feedback controller demo 6-15
  and FixPt Look-Up Table (2D) block 10-151
  and percent safety margin 9-4, 9-29
  interface 9-28
  script 9-4

B

back propagation
  Data Type Propagation block 10-49
  FixPt Gateway Out block 10-107
  FixPt Tapped Delay block 10-197
base data type 7-3
  and feedback controller demo 6-8
binary point 3-3
bit
  clear 10-32
  hidden 3-18
  mask 10-32
  multipliers 3-7
  set 10-32
  shifts 4-41
bits 3-3
block configuration 2-2
  selecting a data type 2-3
  selecting a scaling 2-5
block icon labels 10-10
block parameters 10-4
blocks
  Abs 10-22
  Accumulator 10-23
  Accumulator Resettable 10-25
  Accumulator Resettable Limited 10-27
  Bit Clear 10-29
  Bit Set 10-30
  Bitwise Operator 10-31
  Compare To Constant 10-35
  Compare To Zero 10-36
  Constant 10-37
  Conversion 4-43, 10-40
  Conversion Inherited 10-42
  Cosine 10-44
  Counter Free 10-45
  Counter Limited 10-46
  Data Type Duplicate 10-47
  Data Type Propagation 10-49
  Dead Zone 10-58
  Dead Zone Dynamic 10-60
  Decrement Real World 10-61
  Decrement Stored Integer 10-62
  Decrement Time To Zero 10-63
  Decrement To Zero 10-64
  Derivative 10-65
  Detect Change 10-67
  Detect Decrease 10-68
  Detect Fall Negative 10-69
  Detect Fall Nonpositive 10-70
  Detect Increase 10-71
  Detect Rise Nonnegative 10-72

Detect Rise Positive 10-73
Difference 10-74
Dot Product 10-76
Filter Direct Form I 10-78
Filter Direct Form I Time Varying 10-81
Filter Direct Form II 10-83
Filter Direct Form II Time Varying 10-85
Filter First Order 10-87
Filter Lead or Lag 10-89
Filter Real Zero 10-91
FIR 4-47, 10-93
Free Counter 10-97
Gain 4-45, 10-97
Gateway In 2-2, 10-100
Gateway In Inherited 10-105
Gateway Out 10-107
Increment Real World 10-110
Increment Stored Integer 10-111
Integer Delay 10-112
Integrator Backward 10-113
Integrator Backward Resettable 10-115
Integrator Backward Resettable Limited 10-117
Integrator Forward 10-120
Integrator Forward Resettable 10-122
Integrator Forward Resettable Limited 10-124
Integrator Trapezoidal 10-127
Integrator Trapezoidal Resettable 10-130
Integrator Trapezoidal Resettable Limited 10-133
Interval Test 10-136
Interval Test Dynamic 10-138
Logical Operator 10-139
Look-Up Table 10-141
Look-Up Table (2-D) 10-150
Look-Up Table Dynamic 10-146
Matrix Gain 3-12, 10-155
MinMax 10-159
Multiport Switch 10-163
Product 4-37, 4-40, 10-166
Rate Limiter 10-169
Rate Limiter Dynamic 10-170
Relational Operator 10-171
Relay 10-173
Repeating Sequence Interpolated 10-176
Repeating Sequence Stair 10-178
Sample Time Multiply 10-181
Saturation 10-184
Saturation Dynamic 10-185
Scaling Strip 10-186
Sign 10-187
Sine 10-188
State-Space 10-189
Sum 4-33, 10-192
Switch 10-195
Tapped Delay 10-197
Unary Minus 10-199
Unit Delay 10-200
Unit Delay Enabled 10-202
Unit Delay Enabled External IC 10-204
Unit Delay Enabled Resettable 10-206
Unit Delay Enabled Resettable External IC 10-208
Unit Delay External IC 10-210
Unit Delay Resettable 10-211
Unit Delay Resettable External IC 10-213
Unit Delay With Preview Enabled 10-215
Unit Delay With Preview Enabled Resettable 10-217
Unit Delay With Preview Enabled Resettable External RV 10-219
Unit Delay With Preview Resettable 10-222

Unit Delay With Preview Resettable External RV 10-224
Wrap To Zero 10-226
Zero-Order Hold 10-227
blockset library 10-12
Bode plot 6-6
boolean operation
logical 10-139
broken links 9-24
built-in data types 1-13

C

`ceil` 4-7
chopping 4-9
clearing bits 10-32
code generation 2-8, A-2
and multiplication 4-36
and scaling 10-104
and signal conversions 4-29
and stored integer output 10-107
and summation 4-32
computational noise 4-2
and rounding 4-3
computational units 4-30
constant scaling for best precision 3-12
limitations for code generation A-12
contiguous bits 3-17
conversions
parameter
signal 4-27
See also online conversion, offline conversion
converting old models 9-26

D

data types 2-3, 3-10
built-in 1-13
display 10-10
fractional numbers 2-4
generalized fixed-point numbers 2-4
IEEE numbers 2-4
inherited 10-5
integers 2-3
overriding with doubles 10-9
propagation 10-49
selecting 10-4
dead zone 10-58
demos 2-15
denormalized numbers 3-22
development cycle 1-12
dialog box parameters 10-4
data type 10-4
lock output scaling 10-8
logging min/max data 10-9
overflow handling 10-9
overriding with doubles 10-9
rounding 10-8
scaling 10-6
digital controller 6-7
digital filter 5-2
direct form realization 5-4
and feedback controller demo 6-8
division 10-166
blockset rules 4-39
scaling recommendations 4-23, 4-24
using slope/bias encoding 4-23
dot product 10-76
double bits 4-35, 7-3
double-precision format 3-19
doubles override 9-29

E

Embedded-C Real-Time Target A-8
encapsulation 9-9
encoding scheme 3-5
eps 3-21
examples
  constant scaling for best precision 3-12
  conversions and arithmetic operations 4-46
  converting a built-in model to fixed-point 9-10
  converting from doubles to fixed-point 2-10
  division process 4-40
  fixed-point format 3-7
  fixed-point scaling 3-10
  FixPt Bitwise Operator 10-34
  FixPt FIR 10-96
  FixPt Gateway In 10-102
  FixPt Look-Up Table 10-144
  FixPt Look-Up Table (2D) 10-153
  generating pure integer code A-5
  limitations on precision and errors 4-10
  limitations on range 4-15
  maximizing precision 4-11
  multiplication process 4-37
  saturation and wrapping 4-13
  selecting a measurement scale 1-4
  shifting bits and the radix point 4-42
  shifting bits but not the radix point 4-44
  summation process 4-32
exceptional arithmetic 3-22
exponent for IEEE numbers 3-18
external mode A-12

F

feedback design 6-3
filter
  digital 5-2
fix 4-4
fixed-point interface tool 9-28
  and feedback controller demo 6-9
fixed-point numbers
  general format 3-3
  scaling 3-5
fixpt_convert 9-8
fixpt_convert_prep 9-13
fixptbestexp 9-6
fixptbestprec 9-7
FixPtSimRanges 10-9
float 9-24, 9-25
floating-point numbers 3-18
floor 4-8
fpupdate 9-26
fraction for IEEE numbers 3-18
fractional numbers 2-4
  and guard bits 4-15
fractional slope 3-5
frame-based signals 1-17
fxptdlg 9-28

G

gain 10-97
  matrix gain 10-155
  scaling recommendations 4-22, 4-23
  using slope/bias encoding 4-21
gateway
  fixed-point to Simulink 10-107
  Simulink to fixed-point 10-100
  Simulink to fixed-point, inherited 10-105
generalized fixed-point numbers 2-4
Generic Real-Time Target A-5
global override with doubles 6-12
guard bits 4-15, 9-32, 9-36

GUI
block 9-28
See also fixed-point interface tool

H

help 1-18
hidden bit 3-18

I

icon labels 10-10
IEEE floating-point numbers
format
double precision 3-19
exponent 3-18
fraction 3-18
nonstandard 3-20
sign bit 3-18
single precision 3-19
precision 3-21
range 3-20
infinity 3-23, 4-12
inherited
data types 10-5
by back-propagation 10-49
scaling 10-7
by back-propagation 10-49
Simulink to fixed-point conversion 10-105
installation xvii
integer delay 10-112
integers 2-3
and code generation A-5
outputting large values 10-107
interface

L

least significant bit 3-3
library 10-12
limit cycles 4-2
and feedback controller demo 6-17
lock output scaling 10-8
and feedback controller demo 6-17
logging
large integer values 10-107
overflows 10-9
simulation results 9-29, 10-9
logical operation 10-139
logical shift 4-42
look-up table
1-D 10-141
2-D 10-150
LSB. *See* least significant bit

M

MAC's 4-30
propagating data type information for 10-54
masking bits 10-32
matrix gain 10-155
matrix signals 1-17
maximum value 10-159
logging 10-9
measurement scales 1-2
`mex` xiii
minimum value 10-159
logging 10-9
modeling the system 1-12
most significant bit 3-3
MSB 3-3
multiplication 10-166
blockset rules 4-35
scaling recommendations 4-20, 4-21

using slope/bias encoding 4-20
multiport switch 10-163

N

NaNs 3-23, 4-12
nonstandard IEEE format 3-20

O

offline conversion 4-27
for addition and subtraction 4-31
for multiplication 4-36
for signals 4-28
online conversion
for addition and subtraction 4-31
for multiplication 4-36
for signals 4-28
online help 1-18
overflow 3-21, 4-2, 4-12
and code generation A-4
handling by fixed-point blocks 10-9
logging 6-10, 10-9
overriding with doubles 9-29, 10-9
global override 6-12
individual override 6-18

P

padding with trailing zeros 4-9
and feedback controller demo 6-7
parallel form realization 5-10
parameter conversion 4-27
See also conversions
percent safety margin 9-29
plot system interface 9-30
port data type display 10-10
precision
best 9-6
maximum 9-7
of fixed-point numbers 3-9
of IEEE floating-point numbers 3-21
prerequisites xviii
propagation of data types 10-49

Q

quantization 4-2
and feedback controller demo 6-12
and rounding 4-3
of a real-world value 2-12, 3-6

R

radix point 3-3
radix point-only scaling 3-6
range
of fixed-point numbers 3-9
of IEEE floating-point numbers 3-20
`RangeFactor` 9-4
rapid simulation target A-12
realizations
and data types 7-2
and scaling 7-2
design constraints 5-2
direct form 5-4
parallel form 5-10
series cascade form 5-7
Real-Time Workshop
ERT A-8
external mode A-12
GRT A-5
Production Coder A-5
rapid simulation target A-12

Target Language Compiler A-13
real-world value 3-5
as block input 10-100
relational operation 10-171
relay 10-173
release information 1-18
restoring broken links 9-24
round 4-6
rounding modes 4-3, 10-8
and code generation A-3
toward ceiling 4-6
toward floor 4-7
toward nearest 4-5
toward zero 4-4
rsim A-12
RTW Production Coder A-5

S

saturation 4-13, 10-184
and feedback controller demo 6-10
scaling 2-5, 10-6
and accumulation 4-19
and addition 4-16
and code generation A-4
and division 4-23
and gain 4-21
and multiplication 4-20
constant scaling for best precision 3-12
inherited 10-7
locking 10-8
radix point-only 2-6, 3-6
slope/bias 2-6, 3-6
scientific notation 3-16
series cascade form realization 5-7
setting bits 10-32
sfix 9-31
sfrac 9-32
shifts 4-41
using the FixPt Conversion block 4-42
using the FixPt Gain block 4-43
showfixptsimranges 9-33, 10-9
sign
extension 4-15
of input signal 10-187
sign bit for IEEE numbers 3-18
signal conversions 4-27
Simulink
built-in data types 1-13
converting built-in data types to fixed-point 10-100
converting built-in models to fixed-point 9-8
converting fixed-point data types to built-in 10-107
Simulink Accelerator A-11
single-precision format 3-19
sint 9-34
slope/bias scaling 3-6
stored integer 2-3
as block input 10-100
as block output 10-107
subtraction 10-192
See also addition
switch 10-195
multiport 10-163

T

tapped delay 10-197
Target Language Compiler A-13
targeting an embedded processor 7-3
design rules 7-4
operation assumptions 7-3
size assumptions 7-3

TLC file A-13
truncation 4-9
two's complement 3-3
typographical conventions xviii
typographical conventions (table) xviii

U

ufix 9-35
ufrac 9-36
uint 9-37
unary minus 10-199
underflow 3-21
unit delay 10-200
updating old models 9-26

W

wrapping 4-13

Z

zero order hold 10-227